

Getting started

Tuesday, November 18, 2025 9:39 AM

Below is a **deep, structured, theory-level explanation** of an **Ethernet driver in Linux**, *layer by layer*, exactly the way it is implemented inside the kernel.

I will cover:

1. What an Ethernet driver actually is
2. How packets travel through the driver
3. All important kernel structures (`net_device`, `sk_buff`, NAPI, DMA descriptors, PHY, MDIO)
4. TX path (transmit path) in detail
5. RX path (receive path) in detail
6. Initialization: `probe()`, `open()`, `stop()`
7. Interrupt handling (IRQ)
8. Link negotiation (PHY)
9. Memory management
10. Practical architecture used in SoCs (AM335x, STM32, etc.)

1. What an Ethernet Driver Actually Is

An **Ethernet driver** is the kernel module responsible for connecting the **Linux networking stack** to **hardware Ethernet MAC + PHY**.

When an application does:

```
sendto(sock, buffer, len, ...);
```

Linux converts it into a **`sk_buff`** and hands it to the Ethernet driver.

Your driver:

- allocates DMA descriptors
- copies packet into DMA-able memory
- sets hardware registers
- handles interrupts
- takes incoming packets from DMA
- passes them to the kernel networking stack
- does link negotiation with the PHY

Ethernet driver = **Bridge between software networking stack and physical wire**.

2. Linux Networking Stack Layering

User-space app (socket)

↓
TCP/UDP layer

↓
IP layer

↓
ARP (for IPv4)

↓
Network device (Ethernet driver)

↓
MAC Controller (Ethernet controller in SoC)

PHY (Physical layer chip)



RJ45 connector

The driver sits **between networking stack and MAC**.

3. Important Kernel Structures

A. The struct net_device

This represents one network device (eth0, eth1).

Driver must allocate and register it:

```
struct net_device *ndev = alloc_etherdev(sizeof(private_data));
```

Inside net_device:

- **ndo_open** → called on ifconfig up
- **ndo_stop** → called on ifconfig down
- **ndo_start_xmit** → transmit packet
- **ndo_set_rx_mode** → multicast filtering
- **ndo_do_ioctl** → for custom driver ioctl
- **MAC address**
- **MTU**
- **Statistics**

B. struct sk_buff (SKB)

This is the packet container used by Linux networking stack.

Contains:

- data pointer
- len
- protocol
- checksum info
- socket info

The driver receives an **skb** for TX and allocates one for RX.

C. DMA Descriptor Rings (TX & RX Rings)

Most Ethernet MACs use **circular DMA descriptor rings**.

Each descriptor contains:

- Address of packet buffer
- Length
- Status bits
- OWN bit (owned by DMA or CPU)

Example (simplified):

```
struct dma_desc {  
    uint32_t addr;  
    uint32_t len;  
    uint32_t status;  
};
```

D. NAPI (New API for networking)

Modern Ethernet drivers use **NAPI** for high-performance polling.

Goal: reduce interrupt storm during high traffic.

NAPI workflow:

11. Packet arrives → hardware triggers interrupt
12. Driver disables further RX interrupts
13. Schedules **poll()** function
14. poll() pulls packets from DMA ring (up to a budget)
15. When ring empty → re-enable RX interrupts

E. PHY & MDIO

PHY = physical layer chip (10/100/1000 Mbps)

MAC ↔ PHY communication uses **MDIO bus**.

Driver must:

- read PHY status (link up/down)
- read/write registers
- handle auto-negotiation
- handle link speed and duplex mode changes

Linux uses **phylib**, and your driver registers:

```
phydev = phy_connect(ndev, phy_id, &adjust_link, 0);
```

4. TX Path in DEAD DETAIL

TX starts from user-space → skb → hardware.

Step-by-step

1. Application writes packet using socket()

2. Networking stack processes it

Adds headers: IP → Ethernet → etc.

3. Calls driver's:

```
ndo_start_xmit(struct sk_buff *skb, struct net_device *ndev)
```

4. Driver does:

- select next free TX DMA descriptor
- map skb->data for DMA using:
dma_map_single()

- fill descriptor:

- buffer address

- length

- set OWN bit

5. Kick hardware

Write to MAC registers to start DMA:

```
TX_POLL_DEMAND = 1
```

6. Hardware sends the packet out

MAC → PHY → Ethernet cable.

7. TX Completion Interrupt

ISR wakes driver:

- checks which descriptors are done
- unmaps DMA
- frees skb
- updates statistics

5. RX Path in DEAD DETAIL

RX path is more complex.

1. At initialization

Driver allocates many RX buffers and RX descriptors:

```
skb = netdev_alloc_skb()
dma_map_single()
desc->addr = dma_addr
desc->OWN = 1
```

2. Packet arrives

PHY → MAC → DMA writes packet into buffer.

Descriptor OWN bit cleared (DMA to driver).

3. RX interrupt fired

ISR disables further interrupts

Schedules NAPI poll()

4. NAPI poll() runs

Steps inside poll():

while (budget > 0 && RX descriptor has packet):

```
    unmap DMA buffer
    skb_put() length
    set skb->protocol
    pass skb upward using netif_receive_skb()
    allocate new skb for RX ring
    map it for DMA
    give descriptor back to hardware (OWN=1)
```

5. When RX ring empty

Enable interrupts again.

6. Initialization Workflow

All Ethernet drivers follow this sequence:

Probe

Called when driver matched to device (DTB or PCI):

```
probe()
{
    alloc net_device
    ioremap registers
    alloprobe()
}
```

```

alloc net_device
ioremap registers
allocate DMA rings
init NAPI
register net_device
}
cate DMA rings
init NAPI
register net_device
}

```

[Open](#)

Triggered by:

ifconfig eth0 up

Open must:

- request IRQ
- start TX/RX DMA
- connect to PHY
- start PHY auto-negotiation
- enable MAC

[Stop](#)

Triggered by:

ifconfig eth0 down

Stop must:

- stop TX/RX DMA
- free IRQ
- stop PHY
- disable MAC

[7. Interrupt Handling in Ethernet Driver](#)

Interrupt types:

16. **RX interrupt** → packet received
17. **TX interrupt** → packet transmitted
18. **Link change interrupt** → link up / link down
19. **Error interrupt**

ISR should be short:

```

irq_handler()
{
    disable RX interrupt
    napi_schedule()
}

```

Long work is done in NAPI poll().

8. PHY Link Handling in Detail

Linux phylib handles:

- auto-negotiation (speed, duplex)
- cable connect/disconnect
- pause frame support
- reading PHY registers (status, control, ID)

Your driver provides:

```
void adjust_link(struct net_device *ndev)
{
    if (phydev->link)
        mac_config_speed_duplex();
    else
        stop_tx_rx();
}
```

9. Memory & DMA Theory

Ethernet driver uses **DMA-able memory**.

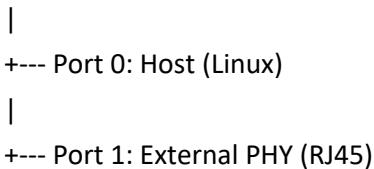
Rules:

- Cannot use normal kmalloc() for RX/TX buffers
- Must use consistent memory or dma_alloc_coherent()
- Must align buffers (2-byte, 4-byte, 8-byte aligned depending on MAC)
- Must handle cache coherency (ARM needs cache maintenance)

10. Real-World Example Architecture (BeagleBone AM335x)

AM335x Ethernet subsystem:

CPSW Switch Core



Driver files:

`drivers/net/ethernet/ti/cpsw.c`

Components:

- ALE (address lookup engine)
- CPDMA (DMA engine)
- Descriptor rings
- Slave ports
- MDIO driver
- PHY drivers

Uses NAPI and DMA rings heavily.

If you want, I can also explain:

- ✓ Full CPSW driver architecture of AM335x
- ✓ How DMA descriptors look in detail (bit-by-bit)
- ✓ How PHY registers work
- ✓ How to write a minimal Ethernet driver from scratch
- ✓ How TX/RX rings are allocated
- ✓ How zero-copy works
- ✓ How checksum offloading works
- ✓ How VLAN filtering works

Just tell me **which part you want next.**

Below is a **crystal-clear, step-by-step explanation** of:

20. **What components you must understand before writing an Ethernet driver**
21. **What minimal pieces a simple Linux Ethernet driver must contain**
22. **A super-minimal, educational-only dummy Ethernet driver (no hardware)** — the smallest conceptual one

This will help you fully understand how real drivers (like **cpsw** on BeagleBone) work.

1. BEFORE WRITING ANY ETHERNET DRIVER — WHAT MUST YOU UNDERSTAND

To write even a simple Ethernet driver, you must understand 7 fundamental concepts:

(1) net_device structure

Every network interface in Linux (eth0, wlan0...) is represented by struct net_device.

Your driver *allocates, initializes, and registers* this object.

Important fields:

- device name (ethX)
- MAC address
- MTU
- pointers to your driver's functions (net_device_ops)

(2) net_device_ops (your driver API)

These are your function callbacks:

```
ndo_open      -> start hardware / enable irq
ndo_stop     -> stop hardware
ndo_start_xmit -> send packet
ndo_set_rx_mode -> multicast handling
ndo_tx_timeout -> handle stuck TX
```

In real drivers there are many more (XDP, VLAN...), but simple driver needs only:

- .ndo_open
- .ndo_stop
- .ndo_start_xmit

(3) SKB (socket buffer) — the packet container

Linux passes packets to the driver in **struct sk_buff**

```
ndo_start_xmit(struct sk_buff *skb, struct net_device *dev)
→ kernel gives you a packet to send.
```

Inside skb you have:

- data pointer
- length
- protocol
- checksum info
- headroom/tailroom

(4) RX and TX Rings / Queues

Real NICs use DMA rings:

TX Ring

- array of descriptors
- each descriptor points to packet buffer
- NIC takes descriptor and sends packet

RX Ring

- array of empty buffers
- NIC fills them with incoming packets
- driver converts to SKB and hands to kernel

For minimal dummy driver → rings are not required.

(5) Interrupts

Real drivers use interrupts:

- TX complete interrupt → free descriptor
- RX interrupt → new packet arrived
- Link change interrupt

Dummy driver → we skip IRQs.

(6) NAPI (New API)

Real RX implementation uses:

- interrupt → schedule NAPI poll
- poll function drains RX ring
- improves performance

Dummy driver → we skip NAPI.

(7) PHY (Ethernet PHY layer)

Real drivers must manage:

- PHY auto-negotiation
- speed (10/100/1000)
- duplex mode
- link state

Dummy driver → no PHY.

✓ So for the simplest Ethernet driver, we only need these:

- register struct net_device
- implement .ndo_open
- implement .ndo_stop
- implement .ndo_start_xmit
- simulate TX
- simulate RX using a timer or workqueue

2. SUPER SIMPLE ETHERNET DRIVER (DUMMY DRIVER)

This driver does not talk to hardware.

It simply:

- Pretends to send packets
- Periodically injects fake packets into the kernel

This teaches you the driver structure *without rings, DMA, PHY, hardware.*

Cpsw.C

Wednesday, December 3, 2025 10:59 AM

Great — this is an **excellent place** to understand how a TI CPSW Ethernet driver is registered in Linux. Let's break it down **line-by-line, structure-by-structure, member-by-member**, and understand **why there is no “init function” here.**

1. SIMPLE_DEV_PM_OPS(cpsw_pm_ops, cpsw_suspend, cpsw_resume);

What is this?

This is a **macro** that creates a **power-management ops structure** for the driver.

Expanded meaning

SIMPLE_DEV_PM_OPS(name, suspend_fn, resume_fn) expands to:

```
const struct dev_pm_ops cpsw_pm_ops = {  
    .suspend = cpsw_suspend,  
    .resume = cpsw_resume,  
};
```

What this does in the driver?

This tells Linux:

- When system is **suspending**, call `cpsw_suspend()`
- When system is **resuming**, call `cpsw_resume()`

Used for:

- Turning off CPSW clocks
- Disabling DMA
- Power-saving modes
- Reinitializing hardware on resume

2. static struct platform_driver cpsw_driver = { ... };

This is the **core** of every platform driver.

It tells Linux:

“Hey kernel, I am a driver for a hardware device described in Device Tree.”

Breakdown:

2.1 Outer structure → struct platform_driver

Defined in:

include/linux/platform_device.h

This registers a driver that works via:

- **Device Tree (DT)**

- **platform bus**
- **non-discoverable hardware**

CPSW hardware on TI SoCs is **non-enumerable**, so platform driver is used.

↙ Let's break inside:

```
.driver = {
    .name    = "cpsw",
    .pm      = &cpsw_pm_ops,
    .of_match_table = cpsw_of_mtable,
},
```

◊ .name = "cpsw"

This is the **name of the driver**.

Linux uses this name for:

- /sys/bus/platform/drivers/cpsw
- Debug messages
- Matching legacy non-DT drivers (rare now)

◊ .pm = &cpsw_pm_ops

This attaches the **power management callbacks** created earlier:

```
cpsw_suspend()  
cpsw_resume()
```

Kernel will call these when device is suspended/resumed.

◊ .of_match_table = cpsw_of_mtable

This tells Linux:

“Check Device Tree and match any node that matches entries inside cpsw_of_mtable”.

Typically:

```
static const struct of_device_id cpsw_of_mtable[] = {  
    { .compatible = "ti,cpsw" },  
    { /* sentinel */ }  
};
```

This means:

If Device Tree has something like:

```
ethernet@4a100000 {  
    compatible = "ti,cpsw";
```

}

→ Bind this driver to that hardware.

🔗 Next fields:

.probe = csw_probe,
.remove = csw_remove,

These are CRITICAL.

3. .probe = csw_probe

The **probe function IS the driver init function.**

You asked:

“here no init functions are there?”

This **IS THE INIT FUNCTION.**

Platform drivers **do not use**:

module_init() → init_module() like char drivers.

Instead:

The init sequence is:

1. Device Tree parsed
2. Platform device created in kernel
3. Kernel matches your .of_match_table
4. Kernel calls .probe
5. Driver initialization happens inside csw_probe()

Probe does ALL the following:

Inside csw_probe() you will find:

- request and map registers (devm_ioremap_resource)
- get clocks (clk_get)
- allocate net_device
- allocate private structure
- initialize DMA
- register NAPI
- setup IRQs
- register netdev (register_netdev)

This is the **main init function.**

You will see in your TI driver:

```
static int csw_probe(struct platform_device *pdev)
{
    // entire init logic here
}
```

⚡ 4. .remove = cpsw_remove

This is called when:

- driver is unloaded
- device removed
- system shutdown

Undo everything from probe:

- unregister netdev
- free DMA
- free IRQs
- disable clocks

⚡ SO WHERE IS MODULE_INIT() ???

At the bottom of the kernel file you will find:

```
module_platform_driver(cpsw_driver);
```

This macro expands to:

```
module_init(cpsw_driver_init);  
module_exit(cpsw_driver_exit);
```

And:

```
static int cpsw_driver_init(void)  
{  
    return platform_driver_register(&cpsw_driver);  
}
```

```
static void cpsw_driver_exit(void)  
{  
    platform_driver_unregister(&cpsw_driver);  
}
```

So the real init happens like this:

1. module_init() → registers platform_driver
2. Platform bus finds matching DT node
3. Calls csw_probe()
4. Your NIC is initialized

❖ Summary (in super simple steps)

✓ Step 0: module_platform_driver registers the driver

- ✓ Step 1: Kernel reads Device Tree
- ✓ Step 2: Kernel sees compatible = "ti,cpsw"
- ✓ Step 3: Kernel calls `cpsw_driver.probe = csw_probe`
- ✓ Step 4: `csw_probe` initializes the NIC
- alloc netdev
- init DMA
- init interrupts
- register netdev
- PHY init
- NAPI init
- clock enable
- start CPSW subsystem
- ✓ Step 5: When NIC is opened → `ndo_open()`
- ✓ Step 6: When packet TX → `ndo_start_xmit()`
- ✓ Step 7: When interrupt → ISR → NAPI poll
- ✓ Step 8: When module removed → `csw_remove()`

Variable	Type	Purpose
<code>dev</code>	<code>struct device *</code>	Kernel device object for this platform dev
<code>clk</code>	<code>struct clk *</code>	Handle to CPSW hardware clock
<code>data</code>	<code>struct csw_platform_data *</code>	SoC-specific CPSW configuration
<code>ndev</code>	<code>struct net_device *</code>	Network interface (eth0/eth1)
<code>priv</code>	<code>struct csw_priv *</code>	Private per-netdev data
<code>ss_regs</code>	<code>void __iomem *</code>	Memory-mapped CPSW registers
<code>ss_res</code>	<code>struct resource *</code>	Register range info from DT
<code>mode</code>	<code>struct gpio_descs *</code>	Mode-select GPIO pins
<code>soc</code>	<code>soc_device_attribute *</code>	SoC-specific quirks handler
<code>csw</code>	<code>struct csw_common *</code>	Shared CPSW hardware context

Let's explain `devm_kzalloc()` in a *simple, beginner-friendly, scenario-based* way — **no code**, only concept.

`devm_kzalloc(dev, size, GFP_KERNEL)` — What it does (EXACTLY)

1. It allocates memory

You tell the kernel:

"Please give me a block of memory of size `sizeof(struct csw_common)` so I can store my driver's data."

So the kernel gives you a memory region.

2. The memory is ZEROED

"`kzalloc`" means **kernel zero alloc**.

Every byte is set to **0**.

This is useful because:

- All pointers inside become NULL
- All counters become 0
- No garbage values

So your `cpsw_common` structure starts in a clean state.

3. The memory is "device-managed"

This is the MOST IMPORTANT part.

`devm_` = *device-managed* allocation.

This means:

 The memory will be automatically freed

When will it free?

 When the device is

- removed,
- driver unbound,
- or probe failed.

The kernel automatically cleans the memory.

Why is this useful?

Normally in drivers, you must manually free memory in:

- `remove()`
- or if `probe()` fails somewhere

With `devm_kzalloc()`

 You don't need to free it. Ever.

Kernel cleans it on unbind/remove.

Scenario-based example (VERY SIMPLE)

 Imagine you are opening a restaurant branch.

You need:

- kitchen tools
- tables
- chairs
- cooking equipment

You ask the mall owner:

"Give me all the equipment for this shop."

That's `devm_kzalloc()`.

Now the mall tells you:

"When you close this restaurant,

I will automatically take all the equipment back."

So:

- You don't need to track which tool you took
- You don't need to clean anything
- You don't need to remember to return anything

Everything will be cleaned up for you.

That is **device-managed memory**.

⌚ Why not use simple kzalloc()?

Because then *you* must:

- free memory if probe fails
- free memory in remove
- track every allocation manually

Much harder and error-prone.

⌚ In the CPSW driver's case:

cpsw_common contains:

- registers addresses
- DMA channel pointers
- IRQ info
- PHY pointers
- clocks
- configs
- other state

This memory must live **as long as driver is bound**.

When driver is unloaded or probe fails → kernel frees automatically.

★ FINAL SUMMARY (1 sentence)

devm_kzalloc() allocates zeroed memory that is automatically freed when the device is removed, saving you from writing cleanup code and preventing memory leaks.

Great — let me explain **WHAT A “SLAVE” IS IN TI CPSW**, from **zero level**, with diagrams and real-world examples.

No driver code.

No complex terminology.

Just the concept in simple terms.

❖ What is CPSW? (TI Ethernet Switch)

CPSW = Controller Packet Switch.

It is the Ethernet block inside TI SoCs (AM335x, AM437x, AM57xx, AM62x).

It is **NOT** a simple MAC like on STM32.

It is actually:

A small 3-port Ethernet switch built inside the chip.

Inside TI SoCs:

[Slave 0] [Slave 1]

\ /

\ /

[Internal Switch]

|

[Host Port]

|

(CPU / Linux Kernel)

This is the **MOST IMPORTANT** thing to understand.

💡 So what is a **Slave**?

✓ A “slave” in CPSW = A **MAC port** on the external side

Each slave connects to a **PHY chip** on the board.

Example:

AM335x (BeagleBone Black) has **2 external Ethernet ports** inside the SoC:

Slave 0 → can connect to PHY0

Slave 1 → connects to PHY1

The SoC designers call these **slave ports**.

💡 Why called “slave”?

Because inside CPSW switch:

- The **host port** is the “master” (owned by CPU).
- The external MAC ports are “slaves” (they receive/speak through the switch).

It is a hardware design naming — nothing to do with software threads or slaves.

💡 The 3 CPSW Ports

Port Name	Connected To	Purpose
Host Port	CPU (Linux)	Traffic enters/exits Linux
Slave 0	PHY 0	Actual Ethernet external port
Slave 1	PHY 1	Second Ethernet port

Real Example — BeagleBone Black

On BBB:

- Only **Slave 1** is used.
- Slave 0 pins are not routed to the connector.

So the driver must choose:

```
use slave_port = 1
```

That's why you saw earlier:

```
cpsw_slave_index = cpsw_slave_index_priv;
```

Why driver needs to know which slave is active?

Because it must configure:

- The MAC address
- The PHY connection
- The link state
- The DMA channels
- The interrupts

for **the correct port**.

If board uses Slave 1 but driver configures Slave 0 → **Ethernet will never work**.

Let's explain in HUMAN terms:

Imagine CPSW as a house:

- **Host Port** = front door (CPU entry)
- **Slave 0** = kitchen door
- **Slave 1** = back door

But your **Ethernet cable is plugged into only one door**.

The driver must know **which door**.

Driver creates a net_device (eth0, eth1) for each slave

Depending on board:

- Some boards use **only 1 slave** → only eth0 created
- Some boards use both slaves → create eth0 and eth1

Example on dual port board:

eth0 → Slave 0

eth1 → Slave 1

Why CPSW has MULTIPLE slaves but only ONE host port?

Because it works like an **Ethernet switch**:

- Multiple external ports
- One internal port (to CPU)

That's why features like:

- VLAN
- Port forwarding
- Learning FDB
- Multicast

exist in CPSW hardware.

Practical Example

Suppose your board uses PHY on slave 1.

During probe:

- It reads device-tree
- Sees that Slave 1 PHY is present
- Configures Slave 1 registers
- Creates one netdev (eth0) bound to Slave 1
- Slave 0 is ignored (unused)

Where “slave” appears in the code?

The structure:

```
struct cpsw_slave {
    void __iomem *regs;
    struct net_device *ndev;
    struct phy_device *phy;
    int slave_num;
    ...
}
```

Each slave has:

- Its own registers
- Its own PHY
- Its own MAC address

Summary (Simple)

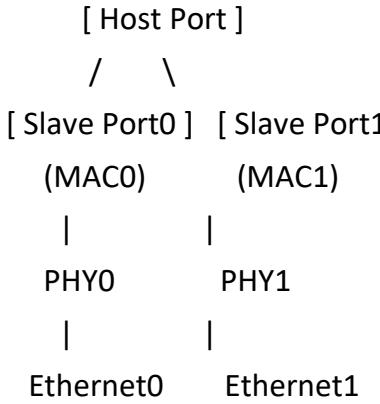
Slaves = Ports that connect to PHY chips → real Ethernet jacks.

Host Port = Internal port → CPU.

CPSW is basically a **3-port switch**:

CPU

|



Yes — **exactly.**

`struct cpsw_common` represents the **entire CPSW hardware block** as one unit:
the **whole internal Ethernet switch** inside the TI SoC.

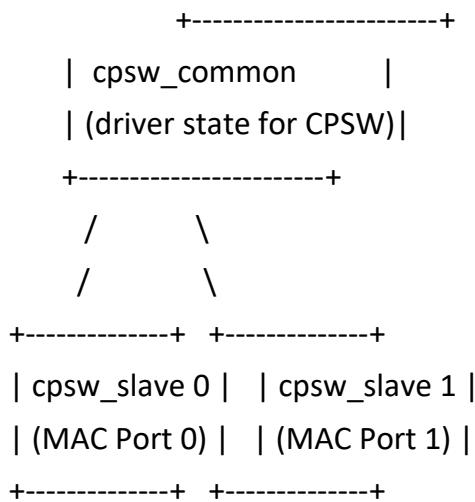
This includes:

- The **host port**
- All **slave ports** (1 or 2 depending on chip)
- Shared switch registers
- Shared DMA subsystem
- Shared IRQs
- Shared ALE (Address Lookup Engine)
- Shared CPTS (time stamping)
- Shared clocks
- Shared resources like MDIO bus

So yes:

cpsw_common = full CPSW switch system inside the chip.

🔍 Visual representation



^

+-----+	
Host Port	
DMA Channels	
Switch Core	
ALE, MDIO	
+-----+	

⌚ What EXACTLY lives inside struct `cpsw_common`?

Let me break it down in simple terms.

✓ 1. Hardware register bases

Pointers to:

- CPSW switch subsystem registers
- Host port registers
- ALE registers
- Slaves' register offsets

✓ 2. Pointers to slaves

`cpsw->slaves[0]` → slave port 0

`cpsw->slaves[1]` → slave port 1

✓ 3. DMA state

- TX/RX DMA channels
- Descriptor rings
- Queue mapping
- Buffer allocation pools

✓ 4. Device-tree / platform data

- MAC addresses
- PHY interface type
- RGMII/MII settings
- Clock frequencies

✓ 5. NAPI objects

Shared NAPI pollers for RX/TX.

✓ 6. Locks & spinlocks

Used for synchronization.

✓ 7. IRQ info

Handles interrupts for:

- RX DMA
- TX DMA
- Link state
- Errors

✓ 8. Clocks

- CPSW clock
- MDIO clock
- CPTS clock

✓ 9. MDIO bus

Used to talk to PHY.

✓ 10. ALE (Address Lookup Engine)

Internal switch forwarding table.

✓ 11. CPTS (Time Sync / PTP)

Used for IEEE1588 timestamping.

 Why not store all this in each interface (eth0, eth1)?

Because:

- The **host port**, DMA, ALE, interrupts, registers are **shared**.
- The hardware is **one switch**, not two MACs.

So `cpsw_common` is the “heart” of the CPSW driver.

Each slave only stores:

- Its PHY
- Its MAC registers
- Its `net_device`
- Its slave-specific configuration

Everything else is in the **shared `cpsw_common`**.

 Simple Analogy

Think of `cpsw_common` as:

The main switch controller that manages the whole Ethernet hardware.

And think of each `cpsw_slave` as:

A wall socket (Ethernet port) connected to the switch.

Yes — **exactly**.

Let me explain **crystal clear**, step-by-step, with BeagleBone Black CPSW architecture in mind.

Does each CPSW slave port use MII/MDIO to communicate with PHY?

YES.

Each **CPSW slave interface** (Slave 0 and Slave 1) connects to an **external PHY chip** using:

- **MII / RMII / RGMII interface** → used to send/receive Ethernet frames
- **MDIO bus** → used to *configure and control* the PHY registers

But the important part:

MDIO is shared

There is **only one MDIO controller** inside the CPSW switch block.

Both slaves use that *same MDIO block* to talk to their own PHYs.

⚙️ CPSW Hardware Block (as used in AM335x / BeagleBone Black)



- **Slave 0** talks to **PHY0**
- **Slave 1** talks to **PHY1**
- Both use the same MDIO controller for config/negotiation

⌚ What is a CPSW Slave in kernel terms?

struct `cpsw_slave` represents **ONE physical MAC port** inside CPSW:

A slave includes:

- Its own MAC port registers
- DMA channels associated with it
- Its own connection to PHY
- Its own struct `net_device` (`eth0, eth1...`)
- Its own interrupt events (RX/TX completion)

Think of a CPSW slave as:

“A separate Ethernet interface exposed by the internal CPSW switch.”

⌚ How each slave interacts with PHY?

Each slave talks to PHY using:

1. MII/RGMII → DATA path

This carries:

- Ethernet frames (TX/RX)

- At 100/1000 Mbps rate

Example:

Slave 0 sends a frame → RGMII → external PHY → RJ45 port → network.

2. MDIO → CONTROL path

Used to operate on PHY registers:

- PHY ID read
- Auto-negotiation start
- Link up/down status
- Speed/duplex configuration
- Reading link partner abilities

Kernel uses MDIO calls like:

```
mdiobus_read()  
mdiobus_write()  
phy_read()  
phy_write()
```

Each slave has:

```
slave->phy (struct phy_device)
```

How driver maps slaves to PHYs

In probe():

- It reads DT “phy-handle = <&phy0>”
- It maps slave → PHY

Example:

```
cpsw->slaves[0].phy_id = 0  
cpsw->slaves[1].phy_id = 1
```

So the driver knows:

- Slave 0 uses PHY ID 0 on MDIO bus
- Slave 1 uses PHY ID 1 on MDIO bus

During open():

```
phy_connect(...)  
phy_start(...)
```

In simple language

Slave = MAC port inside SoC

PHY = chip outside SoC (near RJ45)

MDIO = control wire for PHY

MII/RGMII = data wires to PHY

Slaves → talk to PHY → PHY → talk to RJ45.

☒ Short Answer Again

Yes, CPSW slave ports use MII/RGMII for data transmission and MDIO (shared) for PHY configuration.

Below is the **clear, beginner-friendly, CPSW-specific explanation** of what these two functions actually **do inside the Ethernet driver** — not generic Linux API definitions.

1. `devm_gpiod_get_array_optional(dev, "mode", GPIO_OUT_LOW)`

What it does in the CPSW Ethernet driver?

This function gets a **set/array of GPIO pins** named "mode" from Device Tree.

These GPIOs control the **MAC port mode** (like "MII", "RMII", "RGMII", "Internal delay", etc.) by physically driving pins on the SoC that select MAC/PHY configuration.

Why CPSW needs GPIO mode pins?

TI AM335x/AM437x CPSW MAC can operate in different modes:

- **MII mode**
- **RMII mode**
- **RGMII mode**
- **RGMII with internal delay**
- **PHY interface mode selection**

On these SoCs, these modes may be selected using *strap pins* or GPIO pins.

So the driver needs to **configure these pins** during probe.

What this call actually does:

6. Reads from Device Tree:

```
mode-gpios = <&gpio1 5 GPIO_ACTIVE_HIGH>, <&gpio1 6 GPIO_ACTIVE_HIGH>;
```

7. Requests these GPIO pins
8. Sets them **OUTPUT LOW** (GPIO_OUT_LOW)
9. Used to configure interface mode before driver enables switch ports

Example scenario (CPSW reality)

Assume BeagleBone Black has:

- GPIO1_5 and GPIO1_6 selecting RGMII mode for Slave 1

Driver does:

```
mode = devm_gpiod_get_array_optional(dev, "mode", ...);
```

Then writes correct logic values to pins → PHY and MAC are now in RGMII mode.

Without this:

 MAC may come up in wrong mode

 Ethernet will not work

- ✗ Auto-negotiation may fail
- ✗ No link or corrupted RX/TX

2. `clk = devm_clk_get(dev, "fck")`

What it does *in the CPSW driver?*

This function **gets the functional clock** ("fck") for the CPSW module.

The CPSW Ethernet switch and ports cannot work unless the **clock is enabled**.

✓ CPSW uses several clocks:

- `cpsw_fck` → functional clock to the switch
- `cpts_rft` → CPTS time sync reference
- `mdio_fck` → MDIO controller clock

Here you are fetching:

→ "fck" = Functional clock for CPSW core

The clock is required to:

- ✓ Power the internal switch
- ✓ Power the DMA engine (CPDMA)
- ✓ Power the slave MAC ports
- ✓ Enable MDIO transactions
- ✓ Allow register access

If clock is not enabled:

- ✗ Driver cannot access registers
- ✗ MDIO does not work (PHY cannot be detected)
- ✗ No RX/TX
- ✗ Probe will fail

What happens next?

After fetching:

```
clk = devm_clk_get(dev, "fck");
```

Driver will later do:

```
clk_prepare_enable(clk);
```

This:

- Turns ON CPSW switch
- Enables internal data path
- Enables MDIO
- Makes hardware ready for initialization

TL;DR — Simple summary

`devm_gpiod_get_array_optional("mode")`

→ Gets GPIO pins used to select Ethernet MAC/PHY mode (MII/RMII/RGMII).

→ Driver configures hardware interface mode using these GPIOs.

`devm_clk_get("fck")`

→ Gets CPSW functional clock.

→ Driver enables it so CPSW switch + MDIO + ports work.

Let's go line-by-line and I'll explain **exactly what these do INSIDE THE CPSW ETHERNET DRIVER**, not generic kernel docs.

1. `cpsw->bus_freq_mhz = clk_get_rate(clk) / 1000000;`

 **What happens?**

- You already fetched the CPSW **functional clock** (fck).
- Now you read the actual frequency of that clock from hardware.

Example:

`clk_get_rate(clk)` → 125000000 Hz

That is **125 MHz**.

So they store:

`cpsw->bus_freq_mhz = 125;`

Why does CPSW need its clock frequency?

Because many internal timing parameters depend on bus frequency:

- DMA descriptor pacing
- CPSW internal FIFO timing
- MII / RGMII timings
- ALE aging timer
- CPTS timestamp scaling
- MDIO timeout calculations

So the driver must know this to program the registers correctly.

2. `ss_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);`

 **What this does in CPSW driver?**

TI CPSW hardware has a **register memory region**.

Example from device tree:

`reg = <0x4a100000 0x800>;`

This is the **Switch Subsystem (SS)** register region.

This call finds that region:

- Resource 0 of type IORESOURCE_MEM
- Starting physical address of CPSW registers
- Size of register block

Example result:

```
ss_res->start = 0x4A100000
ss_res->end   = 0x4A1007FF
ss_res->flags = IORESOURCE_MEM
```

3. ss_regs = devm_ioremap_resource(dev, ss_res);

✓ What this does inside CPSW?

This maps the **physical register block** into **kernel virtual memory** so the driver can access it.

Before mapping:

You can't write registers at 0x4A100000 directly.

After mapping:

ss_regs becomes something like:

ss_regs = 0xffff000010000000 (kernel virtual address)

Now the driver can do:

```
writel(value, ss_regs + CPSW_REG_OFFSET);
```

This is how the driver configures everything:

- DMA
- ALE
- Host Port
- Slaves
- Statistics
- MDIO controller

All of CPSW hardware is programmed through these mapped registers.

4. cpxsw->regs = ss_regs;

The CPSW driver stores the mapped register base inside cpxsw_common:

```
cpxsw->regs = ss_regs;
```

Why?

So other parts of the driver can access hardware registers easily:

- init functions
- open()
- stop()
- NAPI RX/TX

- interrupt handler
- MDIO initialization
- PHY link change handler

Example usage:

```
writel(0x1, cpsw->regs + CPSW_SOFT_RESET);
```

Putting it together

Code	What it means for CPSW Driver
clk_get_rate()	Get the switch clock frequency (must know this to configure timing).
platform_get_resource()	Get physical address of CPSW register block.
devm_ioremap_resource()	Map that physical memory into kernel virtual space.
cpsw->regs = ...	Save the mapped address so driver can access registers.

One-line intuition for each:

- **Clock frequency** → How fast the CPSW hardware runs
- **platform resource** → Where the CPSW hardware registers exist
- **ioremap** → Make that hardware accessible to the CPU
- **store regs** → Save pointer to control the switch later

Great — this is an important line because it maps the **WR (Wrapper) register block** of the CPSW switch.
Let's break it down cleanly, clearly, and CPSW-specifically.

Code:

```
cpsw->wr_regs = devm_platform_ioremap_resource(pdev, 1);
```

What does this map?

TI CPSW hardware has **multiple register blocks**, not just one.

Typical CPSW memory layout looks like this:

Index	Register Block	Purpose
0	CPSW Subsystem registers (Switch core, Host port, Slaves, Stats, ALE)	Main switch logic
1	WR (Wrapper) registers	Controls reset, interrupts, DMA, misc switch features
2	MDIO controller registers	PHY access
3	CPSW CPDMA registers	DMA engine

So earlier you mapped **resource 0**:

```
ss_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
ss_regs = devm_ioremap_resource(dev, ss_res);
```

```
cpsw->regs = ss_regs;
```

Now you are mapping **resource 1**:

```
cpsw->wr_regs = devm_platform_ioremap_resource(pdev, 1);
```

💧 So what exactly are WR (Wrapper) registers?

The **Wrapper (WR)** block is a small control module that sits on top of the CPSW switch.

It provides:

✓ 1. Global Soft Reset

The WR has a SOFT_RESET register that resets:

- DMA
- ALE
- Host Port
- Slave Ports

Example:

```
writel(SOFT_RESET, cpsw->wr_regs + CPSW_WR_SOFT_RESET);
```

✓ 2. Interrupt Control

WR block handles:

- RX interrupt enable/mask
- TX interrupt enable/mask
- Misc interrupt status
- DMA channel interrupt routing

The driver uses this to **enable/disable IRQs** during open/close.

✓ 3. Control of statistics block

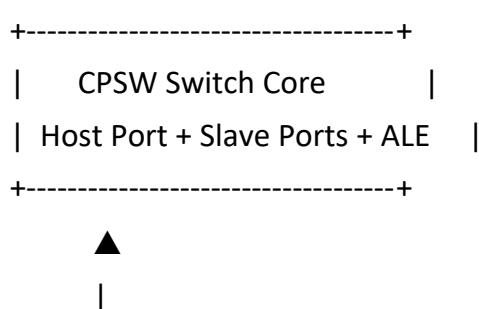
The WR block also controls the switch CPU port statistics logic.

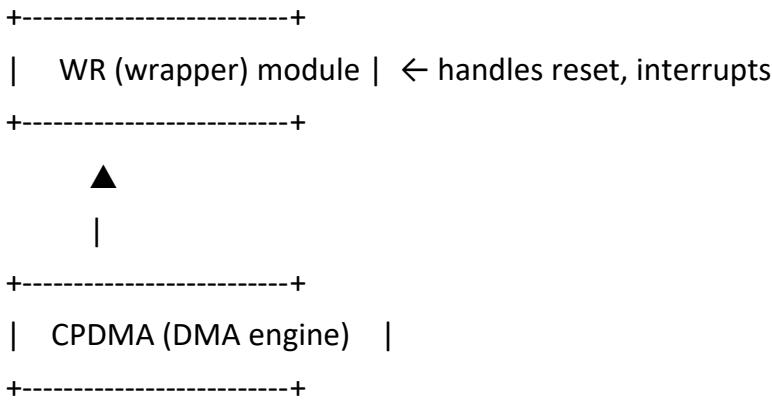
✓ 4. DMA interrupt pacing

WR registers configure how fast interrupts should fire (interrupt moderation).

❖ Why does CPSW need a WR wrapper module?

Think of CPSW hardware like this:





The WR sits above everything and controls them.

Why the driver maps wr_regs separately?

Because:

- WR registers live in a DIFFERENT memory region than CPSW core registers.
- They have different offsets.
- They perform different functions.

You cannot mix them with cpsw->regs.

What does devm_platform_ioremap_resource() do?

Equivalent to:

10. Find the Nth resource (index = 1)
11. Check it's memory type (IORESOURCE_MEM)
12. ioremap() that region
13. Auto-free on driver unload (devm)

Simple.

Example: device tree for CPSW on AM335x

```

reg = <
    0x4a100000 0x800 // resource 0: CPSW core registers
    0x4a101200 0x100 // resource 1: WR wrapper registers
    0x4a101000 0x100 // resource 2: MDIO registers
>;

```

So the call maps the **second** block (index 1).

TL;DR

✓ cspw->wr_regs = pointer to CPSW Wrapper registers

- ✓ Wrapper module = reset controller, interrupt controller, DMA pacing controller
- ✓ Required to start/reset/control the switch hardware

Without WR registers:

- ✗ CPSW cannot be reset
- ✗ Interrupts cannot be enabled
- ✗ DMA cannot be started properly
- ✗ Ethernet would not work at all

Let's break this down **exactly as it works inside CPSW Ethernet driver** — simple, clear, and CPSW-specific.

Full Explanation: CPSW IRQ Fetching

CPSW hardware uses **multiple interrupts**, each responsible for a different part of the switch operation.

These lines fetch those interrupts from **Device Tree resources**.

1. RX IRQ (Receive Interrupt)

```
irq = platform_get_irq(pdev, 1);
if (irq < 0)
    return irq;
cpsw->irqs_table[0] = irq;
```

✓ What this interrupt does?

This is the interrupt generated when **a packet is received** by CPSW DMA.

When a frame arrives:

14. The CPDMA puts the frame into an RX descriptor.
15. RX DMA channel fires IRQ.
16. NAPI is scheduled.
17. Driver drains RX descriptors → gives packets to Linux network stack.

So:

RX IRQ = Packet received interrupt.

Without this → NO data can be received.

2. TX IRQ (Transmit Complete Interrupt)

```
irq = platform_get_irq(pdev, 2);
if (irq < 0)
    return irq;
cpsw->irqs_table[1] = irq;
```

✓ What this interrupt does?

This is fired when a TX DMA descriptor has been successfully transmitted.

Workflow:

18. Kernel gives packet → driver → TX DMA descriptor.
19. DMA transmits it.
20. TX IRQ fires.
21. Driver frees skb, updates stats, wakes queue.

So:

TX IRQ = Packet transmitted acknowledgment.

Without this → TX queues will stop → NO data transmit.

💧 3. Misc IRQ (ALE / LINK / ERROR / STATS / CPTS / etc.)

```
irq = platform_get_irq(pdev, 3);
if (irq <= 0)
    return irq;
cpsz->misc_irq = irq;
```

✓ What this interrupt handles?

This is the **most important multi-purpose IRQ** in CPSW.

It includes:

✓ **ALE (Address Lookup Engine) events**

- ALE table violations
- Unknown packet types
- Multicast flooding events
- Port-based events

✓ **Link State Changes (via IRQ-triggered MDIO)**

- PHY link up
- PHY link down
- Auto-negotiation complete

✓ **Error conditions**

- RX/TX DMA errors
- Port state errors
- FIFO overflows

✓ **Stats interrupts**

- Host port stat block interrupts

✓ **CPTS Timestamp interrupts**

- PTP 1588 timer rollover
- Event capture notifications

So:

Misc IRQ = Link change, ALE events, timestamp events, errors.

Without this the driver wouldn't detect link up/down or errors.

Why 3 separate interrupts?

IRQ	Used For	Why separate?
irq1	RX DMA	High frequency, isolated for NAPI
irq2	TX DMA	High frequency, TX completions
irq3	Misc	Low-frequency but critical events

TI CPSW hardware exposes these as separate IRQ lines to avoid IRQ storms and increase performance.

Where these IRQs come from? (Device Tree)

Example from TI AM335x / AM437x CPSW DTS:

```
interrupts = < 40 41 42 >;  
interrupt-names = "tx", "rx", "misc";
```

Mapped as:

- 0 → TX
- 1 → RX
- 2 → Misc

But in the driver, the order is typically:

- index 1 → RX
- index 2 → TX
- index 3 → MISC

Drivers depend on DT order.

Summary (Very Simple)

Code	Interrupt Type	What it does
platform_get_irq(pdev, 1)	RX IRQ	Packet received
platform_get_irq(pdev, 2)	TX IRQ	Transmit complete
platform_get_irq(pdev, 3)	Misc IRQ	Link change, ALE event, MDIO, CPTS, errors

Great question — you are thinking exactly in the right direction.

Let's break this down very clearly because many beginners get confused by "**polling mode + interrupts**".

FIRST: CPSW DOES NOT RUN 100% IN PURE POLLING MODE

Linux network drivers typically use:

 **Interrupt-driven “wake-up”**

AND

 **NAPI polling for packet processing**

This is called **interrupt moderation + NAPI hybrid mode**.

It is **NOT full polling mode** like a busy loop driver.

So how does it actually work?

✓ Step 1 — Packet arrives → Hardware generates an interrupt

- RX interrupt fires
- Driver disables further RX interrupts
- Schedules NAPI poll() function

This interrupt happens **one time per batch**, not per packet.

✓ Step 2 — NAPI begins polling

Inside NAPI polling:

- Driver repeatedly checks RX descriptors
- Fetches packets
- Processes them

This is the “polling mode.”

But it is **enabled only after an interrupt**, not always running.

✓ Step 3 — When packet queue is empty, NAPI stops polling

When NAPI finishes processing:

- It re-enables RX interrupts
- Then goes idle
- Hardware will interrupt again on next receive event

So polling only happens when needed → **efficient**.

Now your question:

? “If CPSW works in polling mode, why are interrupts needed?”

✓ Because interrupts **wake up** NAPI.

Without interrupts:

- ✗ Driver would never know when to start polling
- ✗ It would have to busy-loop (waste CPU)
- ✗ No packet would be processed unless CPU continuously polls registers

Interrupts act like:

“Hey driver, packets have arrived! Start polling now.”

? When exactly is each interrupt generated?

Let's explain the three interrupts you saw:

💧 1. RX IRQ — when RX DMA fills a descriptor with a packet

Generated when:

- ✓ A frame arrives
- ✓ DMA moves it into an RX descriptor
- ✓ RX channel status bit goes high

Condition: RX_INT set in CPDMA channel interrupt register.

This IRQ is **ONLY to trigger NAPI**.

After that, polling takes over.

💧 2. TX IRQ — when TX DMA finishes sending a packet

Generated when:

- ✓ DMA completed transmission
- ✓ TX descriptor is returned to driver
- ✓ TX complete bit set in CPDMA registers

TX IRQ is **ALWAYS** used (NAPI is not used for TX).

It frees skbs and wakes the netdev queue.

💧 3. MISC IRQ — for special events

Generated when:

- ✓ PHY link goes up/down
- ✓ ALE reports a violation or learning event
- ✓ CPTS timestamp event happens
- ✓ Switch error occurs
- ✓ MDIO interrupt happens

This IRQ is NOT related to RX/TX data path.

🧠 So what is “polling mode” in CPSW?

It refers ONLY to:

- ✓ NAPI RX polling

NOT general polling.

NAPI = interrupt-driven polling.

⚡ SUPER SIMPLE SUMMARY

Phase	Interrupt?	Polling?	Purpose
Packet arrival	Yes (RX IRQ)	No	Kick NAPI to start polling
Packet processing	No	Yes (NAPI polling)	Drain RX DMA descriptors
DMA error, link change	Yes (MISC IRQ)	No	Handle PHY/ALE events

TX completion	Yes (TX IRQ)	No	Free skbs
---------------	---------------------	----	-----------

 If you want I can now explain:

- Exactly which CPSW register bit fires RX IRQ
- How RX IRQ flows into NAPI poll()
- How the driver disables/enables interrupts
- What happens inside cpsw_rx_interrupt()

Just tell me "continue" or paste next lines of probe().

4. What is NAPI busy looping?

NAPI defines **polling with a budget**.

Busy loop meaning:

- The driver enters a loop in poll() to keep pulling packets from RX descriptors.
- It does this **without interrupts** (pure polling) until:
 - Either packets finish
 - Or the poll budget is exhausted

This is called "**coalesced polling**" or "**NAPI busy loop**".

It is not 100% busy wait; it's *intelligent polling*:

- Only happens when there are packets
- Stops when packets finish
- Interrupts resume when queue idle

Let me explain **NAPI busy-looping with a super simple, REALISTIC example**, just like how a NIC driver (such as CPSW) works internally.

I will show:

22. **What happens when packets arrive**
23. **How interrupt wakes NAPI**
24. **How NAPI enters a busy loop**
25. **When NAPI stops and re-enables interrupts**

Using a **story + code + timing diagram** so you visualize it clearly.

Imagine this scenario

Your NIC (CPSW) receives packets at these times:

Packet #	Arrival time
P1	10 ms
P2	10.1 ms
P3	10.2 ms
P4	10.3 ms
No more packets	After 10.3 ms

Assume **NAPI poll budget = 3**.

This matches real Linux NAPI behavior.

Step-by-Step Example

STEP 1 — First packet arrives

At 10 ms:

- RX descriptor fills with packet 1
- NIC generates **1 interrupt** (RX interrupt)

Interrupt handler runs

```
cpsw_disable_rx_interrupt();  
napi_schedule(&priv->napi);
```

It disables interrupts and schedules the NAPI poll.

STEP 2 — NAPI is called, begins busy loop

Driver enters:

```
cpsw_poll(struct napi_struct *napi, int budget = 3)
```

Busy loop starts

```
count = 0;
```

```
while (count < 3) {  
    skb = get_next_rx_packet();  
    if (!skb)  
        break;  
  
    process(skb);  
    count++;  
}
```

 **Packets arriving during polling DO NOT generate interrupts now.**

NAPI busy loop processing (with example packets)

Iteration 1 (at time 10 ms)

```
count = 0 < 3
```

Packet P1 available → processed

```
count = 1
```

Iteration 2 (time 10.1 ms)

P2 arrives *during polling* → immediately picked

count = 2

Iteration 3 (time 10.2 ms)

P3 arrives → processed

count = 3

Iteration 4 (count == budget)

Budget exhausted

Loop ends **even though P4 came at 10.3 ms.**

Because NAPI avoids hogging CPU.

➊ STEP 3 — Budget exhausted → keep NAPI active

Since count = budget, NAPI does **NOT** re-enable interrupts.

Meaning:

- NAPI stays scheduled
- No RX interrupt will fire
- Driver continues polling soon again (soft-IRQ)

Timer or softirq will reschedule NAPI.

➋ STEP 4 — NAPI polls again

poll() runs again:

count = 0

Iteration 1

Packet P4 is waiting → processed

count = 1

Iteration 2

No more packets → get_next_rx_packet() == NULL

loop breaks.

➌ STEP 5 — RX queue empty → re-enable interrupts

Since driver saw “**no packets left**” before hitting budget:

```
napi_complete(napi);  
cpsw_enable_rx_interrupt();
```

➍ Final Result

Time	Event
10.0	P1 arrives → interrupt → NAPI starts

10.1	P2 arrives → picked by active NAPI (no interrupt)
10.2	P3 arrives → picked by active NAPI
10.3	P4 arrives → will be processed in next poll, no interrupt
10.3	NAPI poll cycle ends, queue empty → interrupts re-enabled

Why is this called busy looping?

Because NAPI keeps executing this:

```
while (received < budget && packet exists) {
    process packet
}
```

It continuously loops *as long as packets exist* — but not forever.

It stops when:

- 26. **queue empty**, or
- 27. **budget reached**

This is more efficient than interrupt-per-packet.

Key Points to Understand

Feature	Meaning
Interrupt	Only occurs when RX queue transitions from empty → non-empty
Busy loop	poll() keeps fetching packets until empty or budget reached
No interrupt during polling	Prevents interrupt storms
Interrupt re-enabled when queue empty	CPU sleeps when idle

If you want I can show:

- Simplified CPSW poll code
- Timing diagram with ASCII art
- How to modify CPSW driver to print debug logs inside NAPI
- How TX polling works

Just say what you want next 

Let's break this into **very simple, real-world explanations**.

This part of the CPSW probe() confuses many people because it looks mysterious — but it's actually simple if you understand **Runtime PM (Power Management)**.

Why these lines exist? (Simple Summary)

These lines handle **turning ON** the **CPSW hardware block** so that the driver can safely access registers.

CPSW hardware block = inside AM335x SoC

It gets **powered OFF** by Linux when not used (Runtime PM).

So the driver must:

28. **Enable Runtime PM**
29. **Power ON the CPSW module**
30. **Keep it powered as long as driver is using it**

Now let's explain each line with an easy real-world scenario

Imagine CPSW is a *room with lights off*.

If you want to enter and work inside (read/write registers), you must:

- Turn ON the light → pm_runtime_get_sync()
- When you leave, turn OFF the light → pm_runtime_put()

1. pm_runtime_enable(dev)

```
pm_runtime_enable(dev);
```

Meaning

This tells the Linux kernel:

“This device will use Runtime Power Management. Manage its power states automatically.”

Real-world example

Think of this as telling your apartment building:

“I want automatic lighting and electric control for my room.”

Nothing is powered ON yet — you just **enable the capability**.

Why needed for CPSW?

The CPSW (Ethernet controller) can be:

- Powered ON when network is active
- Powered OFF when unused to save power

So the driver must explicitly enable power management.

2. pm_runtime_get_sync(dev)

```
ret = pm_runtime_get_sync(dev);
```

Meaning

This function:

- ✓ **Powers ON the hardware block**
- ✓ **Enables the clock (fck)**
- ✓ **Makes register access safe**

✓ Waits synchronously until power ON completes

Real-world example

This is like:

“Turn ON the lights in my room so I can work.”

If lights are already ON

Nothing happens (reference count increases).

If lights are OFF

Kernel powers the CPSW module → enables clocks → resets the hardware → then returns.

⚠ What if turning ON fails?

```
if (ret < 0) {  
    pm_runtime_put_noidle(dev);  
    goto clean_runtime_disable_ret;  
}
```

Meaning:

- If hardware failed to power ON
- Or clock failed
- Or something is wrong with SoC PM firmware

Then the driver cleans up and exits.

🧠 Why is this important in CPSW?

Because the driver needs to access registers:

```
cpsw->regs = devm_ioremap_resource()  
cpsw->wr_regs = devm_platform_ioremap_resource()
```

and initialize things like:

- ALE (Address Lookup Engine)
- CPTS (timestamping)
- Slaves / ports
- Switch subsystem registers

If the hardware is still in **low-power mode**, reading registers will crash the kernel.

❖ Putting it all together

Code	Meaning
<code>pm_runtime_enable(dev);</code>	Tell kernel “This device supports dynamic power management.”
<code>pm_runtime_get_sync(dev);</code>	Actually power ON the CPSW hardware block.
<code>pm_runtime_put_noidle(dev);</code>	Undo power ON request if things go wrong.

Small Diagram

Driver loads



Enable Runtime PM



Turn on CPSW clocks & power domain



Now driver can safely access CPSW registers



Continue probe() initialization

Let's explain exactly what this line does, what `cpsw_probe_dt()` is, why it is called here, and what happens if it fails — specifically in context of the CPSW (TI Ethernet Switch) driver.

What is `cpsw_probe_dt()`?

It means:

Parse the Device Tree (DT) and fill `cpsw->data` with all hardware configuration information of CPSW.

DT = Device Tree, which is where the SoC describes:

- How many ports?
- What PHY is connected?
- What MAC addresses?
- What interrupt mapping?
- What is the switch mode?
- What kind of CPSW version (CP(M)AC / CPSW2G / CPSW3G)?
- DMA channels?
- ALE settings?
- Internal or external PHY?
- Pinmux?
- Bus frequencies?
- VLAN / dual-emac mode / switch mode?

The CPSW driver **cannot function** without this information.

So `cpsw_probe_dt()` extracts all this from the DT and puts it inside:

`cpsw->data // struct cpsw_platform_data`

Full meaning of the line

`ret = cpsw_probe_dt(&cpsw->data, pdev);`

It does:

31. Search for the node "cpsw" inside the device tree.
32. Read properties of that node.

33. Store them into `cpsw->data`.

🔍 Why does the driver need `cpsw->data`?

Because CPSW can behave in different modes:

Example scenarios

Scenario 1: BeagleBone Black (Dual EMAC mode)

The board has:

- 2 external PHYs (RGMII)
- 2 MAC addresses
- No switch forwarding (dual-emac mode)

The DT tells the driver:

```
slaves = 2
slave[0].phy_id = 0
slave[1].phy_id = 1
mode = dual_emac
```

Scenario 2: Switch Mode (CPSW as 3-port switch)

Some SoCs run CPSW in switch mode:

```
slaves = 2 ports
ale_enabled = yes
switch_mode = yes
```

Scenario 3: Internal PHY or no PHY

Some boards have *internal PHY*:

```
slave[0].phy-handle = <&internal_phy>;
```

Others use *MII*, *RMII*, *RGMII* modes, each described in DT.

🔍 What exactly is inside `cpsw->data`?

`struct cpsw_platform_data` contains critical configuration:

- number of N/W slave ports
- PHY interface type (MII/RMII/RGMII)
- PHY addresses
- VLAN config
- CPTS (timestamping) enabled?
- ALE (switch engine) enabled?
- dual_emac mode?
- GMII_SEL settings
- hardware version info
- quirks or SoC-specific flags

Without these, CPSW cannot communicate with PHY.

! Why check return value?

```
if (ret)
    goto clean_dt_ret;
```

Because if DT parsing fails:

- Driver does **NOT** know number of ports
- Driver does **NOT** know PHY address
- Driver does **NOT** know interface mode
- Driver might write to **wrong registers**
- Device may not exist / misconfigured

So it must abort probe.

🧠 Simple Real-Life Example

Imagine CPSW is a 3-port Ethernet switch inside the SoC, and DT describes it like:

```
cpsw {
    slaves = 2;
    slave0-phy = PHY at MDIO address 2
    slave1-phy = PHY at MDIO address 3
    mode = "dual_emac";
}
```

cpsw_probe_dt() reads these values and fills:

```
cpsw->data.slaves = 2
cpsw->data.slave[0].phy_id = 2
cpsw->data.slave[1].phy_id = 3
cpsw->data.dual_emac = 1
```

Without this information, CPSW has **no idea where to look for PHY** or how many ports exist.

⚙️ What does clean_dt_ret do?

If DT parsing fails, driver must:

- power-down device (runtime_pm_put / disable clocks)
- free allocated memory
- stop probe sequence

So it jumps to cleanup and exits.

☑️ TL;DR — Easy Version

cpsw_probe_dt() = Read everything about CPSW from the device tree and configure driver.

If DT parsing fails → driver cannot continue → abort.

Great — this is one of the **MOST** important structs in the CPSW driver, and understanding it will unlock the entire driver flow.

I will explain:

- What **struct cpsw_platform_data** contains
- Why **each field** exists
- What **role each plays** in running the CPSW Ethernet MAC
- Real-world examples from BeagleBone Black
- How the driver uses each field

Everything in a **clear, beginner-friendly** way.

Probe dt function in detail

★ FIRST — What is **struct cpsw_platform_data**?

It is a **configuration container** for CPSW.

This struct tells the driver **what kind of CPSW hardware we have** and instructions on how to use it.

Linux fills this struct using:

`cpsw_probe_dt()`

The struct is defined in:

`drivers/net/ethernet/ti/cpsw.h`

★ FULL STRUCT (for reference)

Here is the structure (simplified):

```
struct cpsw_platform_data {  
    u32 slaves;  
    u32 active_slave;  
    u32 cpsw_version;  
  
    struct cpsw_slave_data *slave_data;  
    u32 ale_entries;  
    u32 bd_ram_size;  
  
    bool dual_emac;
```

```
bool switch_mode;  
bool cpts_enabled;  
  
u8 mac_addr[ETH_ALEN];  
u32 host_port_reg_offset;  
  
struct cpsw_ale_params ale_params;  
};
```

Let's explain every field **with examples**.

DETAILED EXPLANATION OF EACH FIELD

1. u32 slaves;

Meaning:

Number of slave ports available on CPSW.

In CPSW:

- CPSW has **2 ports** that connect to PHYs
- Plus **1 host port** (connection to Linux networking stack)

Example (BeagleBone Black):

slaves = 2;

Used by driver to create:

- net_device for eth0
- net_device for eth1

2. u32 active_slave;

Meaning:

Which slave is “active” when in *single* EMAC mode.

If only one Ethernet interface is used, which PHY port?

Example:

If only port 1 is enabled:

active_slave = 0 or 1

Often unused in dual-emac mode.

3. u32 cpsw_version;

Meaning:

Hardware version:

- CPSW2G (AM335x)

- CPSW3G (AM437x)
- CPSW9G (AM57xx)

The driver uses this to apply quirks.

Example:

```
cpsw_version = CPSW_VERSION_2;
```

Because AM335x = CPSW2G.

4. struct csw_slave_data *slave_data;

This is an **array**, one entry per slave port.

slave_data contains:

- PHY address
- PHY connection type (MII / RMII / RGMII)
- Port VLAN config
- GMII selection register offset

Example:

For BeagleBone Black:

```
slave_data[0].phy_id = 0;
slave_data[1].phy_id = 1;
```

These are MDIO bus addresses.

The driver uses this to call:

```
of_phy_connect()
```

5. u32 ale_entries;

ALE = Address Lookup Engine (switch table).

This field tells driver how many entries held by the switch hardware.

Example:

```
ale_entries = 1024;
```

If switch mode enabled, the driver uses this to:

- create forwarding table
- program MAC address filtering
- VLAN filtering

6. u32 bd_ram_size;

Buffer Descriptor RAM size in CPSW.

CPSW has internal SRAM used by TX/RX DMA.

Example:

```
bd_ram_size = 0x2000;
```

Used by the DMA driver to allocate descriptors.

7. `bool dual_emac;`

Meaning:

Enable “dual EMAC mode” = each slave port becomes **its own network interface**.

This is how BeagleBone Black works:

`eth0` = slave 0

`eth1` = slave 1

Why needed?

Because CPSW normally is a *switch* (bridged mode).

Dual-emac turns it into 2 independent MACs.

8. `bool switch_mode;`

Enable 3-port hardware switch behavior:

`eth0 ↔ switch ↔ eth1`

Used in industrial applications where ports forward frames internally.

9. `bool cpts_enabled;`

Enables CPTS = Common Platform Time Sync (IEEE-1588 clock).

Used for:

- PTP time stamping
- Ethernet time sync

Example:

`cpts_enabled = 1;`

10. `u8 mac_addr[ETH_ALEN];`

MAC address for the host port (HW port 0).

Driver copies this to `net_device`.

If not present, driver generates a random one.

11. `u32 host_port_reg_offset;`

Offset to access host port registers inside switch subsystem.

Different SoC revisions place registers differently.

★ How `cpsw_platform_data` is used inside `probe()`

Step 1 — CPSW maps registers

Step 2 — Probe DT

Fills this struct.

Step 3 — Driver configures CPSW hardware:

- reset CPSW
- configure ALE
- configure CPTS
- configure DMA
- configure GMII_SEL
- configure RX/TX channels

Step 4 — Create net devices using data from this struct:

```
cpsw_create_ports();  
register_netdev();
```

★ Real-world Example (BeagleBone Black DT)

DT snippet:

```
ethernet@4a100000 {  
    slaves = <2>;  
    dual_emac = <1>;  
    slave1: ethernet-phy@0 { ... };  
    slave2: ethernet-phy@1 { ... };  
    cpts-en = <1>;  
};
```

Becomes:

```
cpsw->data.slaves = 2  
cpsw->data.dual_emac = true  
cpsw->data.cpts_enabled = true  
slave_data[0].phy_id = 0  
slave_data[1].phy_id = 1
```

⌚ TL;DR — SIMPLE VERSION

cpsw_platform_data describes:

- How many ports CPSW has
- What PHYs are connected
- Mode: dual-emac or switch
- What hardware features exist (CPTS, ALE, versions)
- RAM size
- MAC address

The driver uses this struct to **configure the CPSW hardware and create the network interfaces.**

Okay — now we're stepping into the **Device Tree (DT) parsing part** of the CPSW probe function. You asked specifically about:

```
struct device_node *node = pdev->dev.of_node;
struct device_node *slave_node;
x
```

Let me explain **what they are, why they are needed, and exactly what they do for the CPSW (TI Ethernet Switch) driver**, using clear, real-world examples.

★ 1. What is struct device_node *node?

This is a pointer to the **Device Tree node** representing the CPSW hardware.

In simple words:

node points to the section of the device tree where CPSW is described.

Device tree example (BeagleBone Black):

```
ethernet@4a100000 {
    compatible = "ti,cpsw";
    slaves = <2>;
    dual_emac = <1>;
    ...

    slave0 {
        phy_id = <&davinci_mdio 0>;
        phy-mode = "rmii";
    };

    slave1 {
        phy_id = <&davinci_mdio 1>;
        phy-mode = "rmii";
    };
}
```

When the kernel matches "ti,cpsw" with this node, the platform device gets created.

Then:

```
struct device_node *node = pdev->dev.of_node;
```

node now points to the **ethernet@4a100000** node.

★ Why CPSW uses this node?

Because the driver must read:

- How many slave ports
- What PHY each slave uses
- PHY mode (MII/RMII/RGMII)
- Switch mode or dual-emac mode
- MAC addresses
- CPTS enable flag
- ALE table size
- Child nodes for slaves

So this one pointer (node) gives access to EVERYTHING.

★ 2. What is struct device_node *slave_node?

This is used to iterate through **child nodes** inside the CPSW node.

Example DT:

```
ethernet@4a100000 {  
    slave0 {  
        phy-handle = <&phy0>;  
        phy-mode = "rmii";  
    };  
  
    slave1 {  
        phy-handle = <&phy1>;  
        phy-mode = "rmii";  
    };  
};
```

slave_node will point to these child nodes:

slave_node → "slave0"

slave_node → "slave1"

CPSW has **2 network interface ports**, so:

The driver must read DT for each slave:

- PHY address
- PHY mode
- VLAN tagging
- MAC address override
- MDIO handle

This is done like:

```

for_each_child_of_node(node, slave_node) {
    parse slave0
    parse slave1
}

```

★ Why CPSW MUST parse slave nodes?

Because:

CPSW concept	DT child node
Slave port 0	slave0 DT node
Slave port 1	slave1 DT node

Each port has **its own PHY** with its own MDIO address.

Example:

```

slave0.phy_id = 0
slave1.phy_id = 1

```

If the driver does NOT parse these nodes →

it will NOT know which PHY belongs to which port → network cannot come up.

★ Real-world Example — Let's imagine the driver doing this

DT:

```

ethernet@4a100000 {
    slaves = <2>;
    slave0 {
        phy-handle = <&phy0>;
        phy-mode = "rmii";
    };
    slave1 {
        phy-handle = <&phy1>;
        phy-mode = "rmii";
    };
}

```

Driver:

```

node = pdev->dev.of_node;      // → points to ethernet@4a100000

```

for child in node:

```
slave_node = "slave0"      // first iteration
```

parse phy, mode, etc.

```
slave_node = "slave1"      // second iteration
```

parse phy, mode, etc.

★ How the driver uses slave_node internally?

Example:

```
phy_node = of_parse_phandle(slave_node, "phy-handle", 0);
```

```
phy_id = of_mdio_parse_addr(node, phy_node)
```

and

```
of_get_phy_mode(slave_node)
```

This fills:

```
cpsw->data.slave_data[0]
```

```
cpsw->data.slave_data[1]
```

Used later to create network interfaces.

★ TL;DR — Simple Version

node

→ Points to the CPSW device tree node

→ Contains global CPSW configuration

→ Required for parsing slaves, ALE, CPTS, mode, etc.

slave_node

→ Points to child nodes inside CPSW node

→ Each child represents a CPSW Ethernet port

→ Used to get PHY address, PHY mode, etc.

Let's break this down **like a beginner**, slowly and clearly.

You asked:

What is happening in this code? What is "slaves" property? Why read it? What is data->slaves?

1. The exact code we are understanding

```
if (of_property_read_u32(node, "slaves", &prop)) {  
    dev_err(&pdev->dev, "Missing slaves property in the DT.\n");  
    return -EINVAL;
```

```
}
```

data->slaves = prop;

2. What is being read? → A DeviceTree property

node is the DT node for the CPSW MAC:

```
/* node = pdev->dev.of_node */
```

The code is trying to read:

```
slaves = <2>;
```

from the DeviceTree:

Example CPSW node (BeagleBone Black):

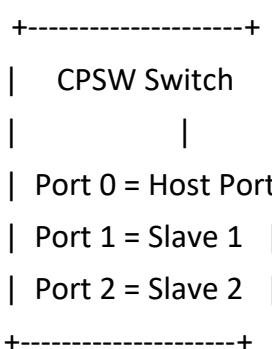
```
cpsw: ethernet@4a100000 {  
    slaves = <2>; // CPSW has 2 slave ports (port 1 and port 2)  
    active_slave = <0>;  
    ...  
}
```

 This “slaves” property tells the driver:

How many MAC ports does the hardware have?

3. Why is "slaves" needed?

TI CPSW (BeagleBone Black Ethernet MAC) architecture:



So total "slaves" = **2** (Port 1 and Port 2).

The driver **must know how many ports to initialize**:

- How many network interfaces to create (eth0, eth1)
- How many PHYs to attach
- How many RX/TX queues to prepare
- How many NAPI structures to create
- How many IRQs to request

If your hardware has only 1 port → slaves = 1

If 2 ports → slaves = 2

4. Step-by-step explanation of the code

Step 1: Try reading DeviceTree property "slaves"

```
if (of_property_read_u32(node, "slaves", &prop)) {
```

- of_property_read_u32 returns 0 on success
- non-zero (true) on failure

So if "slaves" is missing → error.

 This catches DT mistakes.

Step 2: Error printed

```
dev_err(&pdev->dev, "Missing slaves property in the DT.\n");
return -EINVAL;
```

Driver cannot continue → exits probe.

Step 3: Save slaves count

```
data->slaves = prop;
```

Now driver knows how many CPSW MAC ports exist.

5. What is data->slaves?

It is a field of:

```
struct cpsw_platform_data {
    int slaves; // number of slave ports
    ...
}
```

Let's break this code **slowly and clearly** so you understand EXACTLY what happens and WHY.

The code

```
data->slave_data = devm_kcalloc(&pdev->dev,
                                 data->slaves,
                                 sizeof(struct cpsw_slave_data),
                                 GFP_KERNEL);
if (!data->slave_data)
    return -ENOMEM;
```

1. What is being allocated?

The driver allocates an **array** of:

```
struct cpsw_slave_data
```

One entry **per CPSW slave port**.

You already saw:

```
data->slaves = 2;
```

So if slaves = 2:

Allocate array of 2 x struct cpsw_slave_data

This looks like:

Index 0 → slave port 1

Index 1 → slave port 2

2. What is devm_kcalloc()?

It means:

- **Allocate memory** (like kcalloc)
- **Zero-initialize it** (calloc)
- Use **devm_** → the memory will be *automatically freed* when driver is removed
[Why devm?](#)

Because:

- When driver probe **fails** → memory auto-freed
- When driver is unloaded (rmmod) → auto-freed
- You do NOT need kfree()

This prevents memory leaks.

3. Meaning of parameters

```
devm_kcalloc(&pdev->dev, // device that owns memory
```

```
    data->slaves, // number of elements
```

```
    sizeof(struct cpsw_slave_data), // size of each element
```

```
    GFP_KERNEL); // normal kernel allocation flag
```

So memory size =

```
data->slaves * sizeof(struct cpsw_slave_data)
```

If slaves = 2 and struct is 64 bytes:

Alloc = 2 * 64 = 128 bytes

Zero-filled memory returned.

4. What is data->slave_data?

It becomes a pointer to the array:

data->slave_data[0] → info for slave port 1 (eth0)

data->slave_data[1] → info for slave port 2 (eth1)

Later, driver fills this data from DeviceTree.

5. What is stored inside struct cpsw_slave_data?

Something like:

```
struct cpsw_slave_data {  
    u8 mac_addr[ETH_ALEN];  
    int phy_id;  
    char phy_id_name[32];  
    int slave_port;  
    struct device_node *phy_node;  
    struct device_node *slave_node;  
    struct cpsw_slave_platform_data *slave_data;  
};
```

Important fields:

`mac_addr`

- From DT
- MAC for that specific port

`phy_node`

- DT node for PHY connected to that port
(e.g., MDIO address)

 - `slave_node`

- The slave@0 or slave@1 node

 - `phy_id`

- PHY address on MDIO bus

6. Why must we have **one structure per slave port?**

Because each CPSW port has independent:

- PHY
- MAC address
- TX/RX queue mapping

- IRQ mapping
- NAPI context
- Port VLAN ID
- Flow control settings

So the driver needs **per-port configuration**.

Example:

eth0 → uses slave port 1 → MDIO phy = 0

eth1 → uses slave port 2 → MDIO phy = 1

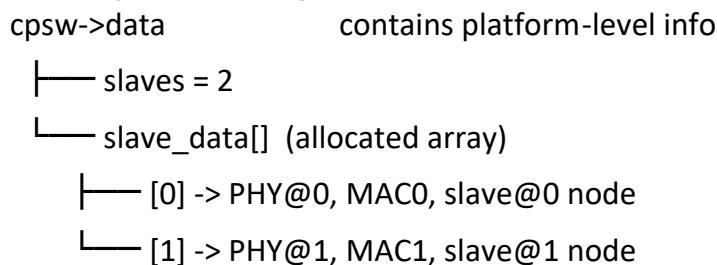
So:

```
data->slave_data[0].phy_id = 0;
```

```
data->slave_data[1].phy_id = 1;
```

Without per-port struct → driver cannot support multiple MAC ports.

Simple visual diagram



Summary

Step	Action
Read DT slaves property	data->slaves = 2
Allocate per-slave array	data->slave_data
Array contains	struct cpsw_slave_data for each port
Why devm_kcalloc	Memory auto-managed
Why needed	Every port has own PHY, MAC, IRQ, NAPI

Let's break this line **very clearly and practically**, because it confuses many beginners:

The code

```
ret = of_platform_populate(node, NULL, NULL, &pdev->dev);

/* We do not want to force this, as in some cases may not have child */
if (ret)
    dev_warn(&pdev->dev, "Doesn't have any child node\n");
```

1. What is of_platform_populate()?

It is a DeviceTree helper that:

Creates platform devices for all child nodes of a given DT node.

For example, if your DeviceTree has:

```
cpsw@4a10000 {
    slaves = <2>

    slave@0 {
        phy-handle = <&phy0>;
    }

    slave@1 {
        phy-handle = <&phy1>;
    }

    mdio@4a101000 {
        #address-cells = <1>;
        #size-cells = <0>;

        phy0: ethernet-phy@0 { ... };
        phy1: ethernet-phy@1 { ... };

    };
};
```

Then

```
of_platform_populate(node, NULL, NULL, &pdev->dev);
```

creates platform devices for:

```
cpsw/slave@0 (child)
cpsw/slave@1 (child)
```

```
cpsw/mdio@4a101000 (child)
cpsw/mdio@4a101000/ethernet-phy@0 (grandchild)
cpsw/mdio@4a101000/ethernet-phy@1 (grandchild)
```

The kernel internally calls:

```
platform_device_register()
```

for each child node.

🔍 2. What this means for CPSW driver?

It means:

The CPSW driver does NOT manually create PHY or MDIO devices.

DT takes care of it.

🔍 3. Why do we need child devices for CPSW?

Because CPSW has internal subsystems:

✓ MDIO controller

→ Manages communication with PHY chips

→ Implemented as a child device: mdio@...

✓ Slave ports

→ Each slave port (port0/port1) has its own config

→ Represented as child nodes: slave@0, slave@1

✓ PHYs under MDIO

→ PHY0 and PHY1 are children of the MDIO node

→ They get platform devices using the same populate call

So the structure in driver becomes:

CPSW (main device)

 |—— slave0_dev (child device)

 |—— slave1_dev (child device)

 |—— mdio_dev (child device)

 |—— phy0_dev (grandchild)

 |—— phy1_dev (grandchild)

⚡ 4. Why does code NOT fail if there are no child nodes?

Because some boards use an older DT layout WITHOUT slave nodes.

Example older BeagleBone DT:

```
cpsw@4a100000 {
    phy_id = <...>;
    mac-address = [....];
};
```

- No slave@0 or slave@1 nodes
- No mdio child
- PHY nodes may exist somewhere else in DT

So the driver handles both DT styles.

Therefore:

```
if (ret)
    warn only, do NOT fail
```

5. Result of the code in the Ethernet driver

On success (children exist):

- MDIO device created
- PHY devices created
- Slave port devices created
- CPSW switches attach to PHYs correctly
- eth0 and eth1 interfaces work

On failure (children missing):

- Driver falls back to legacy binding
- Still works on older systems
- Only a warning printed

Simple Example to Understand

Imagine:

You have a "school" (CPSW device).

Inside school there are:

- Class1 (slave@0)
- Class2 (slave@1)
- Principal office (MDIO)
- Staff (PHY devices inside MDIO)

of_platform_populate() automatically enrolls all classes and staff.

If some kids are missing → school still opens → but you get a warning.

★ Short Summary

Function	Meaning
of_platform_populate()	Create platform devices for all child DT nodes
Why used here?	CPSW has children like slave ports, MDIO controller
Why warning only?	Some older DTs don't define these children

👉 If you want, next I can explain:

- ✓ How the driver finds slave@0 and slave@1 nodes
- ✓ How PHY node is extracted
- ✓ How children device nodes are used by `cpsw_probe_dt`
- ✓ How MDIO device is created and registered

Just say **NEXT**.

Below is a **very clean, extremely detailed beginner-friendly explanation** of the entire code you posted.

I'll explain:

- ✓ why the loop exists
- ✓ what each line does
- ✓ what hardware scenario it represents
- ✓ what each property means (phy-handle, phy_id, fixed-link, mac address, phy-mode, etc.)
- ✓ what the driver is building from this

Let's go step-by-step like a story.

❖ BACKGROUND: What are these "slave ports"?

CPSW = TI's Ethernet Switch.

It has **two** external ports:

`slave@0` → becomes `eth0`

`slave@1` → becomes `eth1`

Each port has:

- Its own PHY chip
- Its own MDIO address
- Its own MAC address
- Its own DT node
- Its own link mode (RGMII/GMII/MII etc.)

The driver must read these details from DeviceTree.

❖ Main loop

```
for_each_available_child_of_node(node, slave_node)
```

This loop walks through ALL child DT nodes of "cpsw".

Example DT:

```
cpsw@4a100000 {
```

```
slave@0 { ... }  
slave@1 { ... }  
mdio@4a101000 { ... }  
}
```

The loop will see:

- slave@0
- slave@1
- mdio@...

But the driver only wants **slave** nodes.

➊ 1. Skip non-slave child nodes

```
if (!of_node_name_eq(slave_node, "slave"))  
    continue;
```

If the node is:

- mdio@... → skip
- random other child → skip

Only process **slave@0 and slave@1**

➋ 2. Select the correct struct to fill

```
struct cpsw_slave_data *slave_data = data->slave_data + i;
```

If $i = 0$ → use array index 0 → slave port 0

If $i = 1$ → use array index 1 → slave port 1

This initializes (fills) the structure for that slave port.

➌ 3. Get port-specific PHY selector cell

```
slave_data->ifphy = devm_of_phy_get(&pdev->dev, slave_node, NULL);
```

Some TI SoCs have a special hardware block that selects which pins go to the PHY.

If enabled, this gives the PHY selector.

If not enabled, it may return an error → ignored unless CONFIG_TI_CPSW_PHY_SEL=y.

➍ 4. Store the slave node pointer

```
slave_data->slave_node = slave_node;
```

Just keeps a reference to DT node for later.

5. Find PHY node: 3 ways

The CPSW driver supports **3 styles** of DeviceTree PHY description.

5A. If DT has phy-handle: (modern, most common)

```
slave_data->phy_node = of_parse_phandle(slave_node, "phy-handle", 0);
```

Example:

```
slave@0 {  
    phy-handle = <&phy0>;  
}
```

Meaning:

- PHY is at MDIO address described in node "phy0"

Driver prints:

```
slave[0] using phy-handle="phy0"
```

5B. If PHY is fixed-link (no external PHY chip)

```
else if (of_phy_is_fixed_link(slave_node)) {  
    of_phy_register_fixed_link(slave_node);  
    slave_data->phy_node = of_node_get(slave_node);  
}
```

Example in DT:

```
slave@0 {  
    fixed-link {  
        speed = <1000>;  
        full-duplex;  
    };  
}
```

Used for:

- RGMII to switch chips
- Internal MAC-to-MAC links
- Embedded PHY inside SoC

5C. If using old style phy_id: (legacy)

```
parp = of_get_property(slave_node, "phy_id", &lennp);
```

Example DT:

```
slave@0 {slave@0 {
    phy_id = <&mdio 1>; // address 1 on MDIO bus
}

    phy_id = <&mdio 1>; // address 1 on MDIO bus
}
```

The driver extracts:

```
mdio_node = of_find_node_by_phandle(parp[0]);
phyid = parp[1];
```

Then builds:

```
slave_data->phy_id = "mdio-name:phy-num"
```

This is VERY OLD — modern DTs do not use this.

5D. If none found → ERROR

No slave[0] phy_id, phy-handle, or fixed-link property

Because the port cannot function without PHY information.

6. Read PHY interface mode (mandatory)

```
ret = of_get_phy_mode(slave_node, &slave_data->phy_if);
```

Examples:

```
phy-mode = "rgmii-id";
phy-mode = "mii";
phy-mode = "rmii";
phy-mode = "sgmii";
```

This tells CPSW hardware which electrical signaling to use.

7. Read MAC address

First try DT:

```
mac_addr = of_get_mac_address(slave_node);
```

If not in DT → read from control module:

```
ti_cm_get_macid(&pdev->dev, i, slave_data->mac_addr);
```

BeagleBone usually uses this method.

8. Dual EMAC: read reserved VLAN

```
data->dual_emac == true
```

Dual EMAC mode → each port becomes totally separate NIC (not bridged).

Read:

```
dual_emac_res_vlan = <123>;
```

If missing, driver assigns:

slave 0 → VLAN 1

slave 1 → VLAN 2

9. Increment port index

```
i++;  
if (i == data->slaves)  
    goto err_node_put;
```

Stops after finishing both slaves.

10. Error cleanup

```
err_node_put:  
    of_node_put(slave_node);  
    return ret;
```

Release the DT node reference.

🧠 💧 Final Summary – What does this entire code actually DO?

It reads all configuration for each CPSW port:

Property	Meaning
phy-handle	Modern way to link MAC ↔ PHY
fixed-link	Used when no external PHY chip
phy_id	Old style PHY reference
phy-mode	Signaling (RGMII, MII, SGMII)
MAC address	Per-port MAC
ifphy	PHY selector cell
VLAN (dual emac)	Per-port VLAN separation

slave_node	Saves DT node
phy_node	Actual PHY node

After this loop, the driver now knows:

- ✓ which PHY is attached to slave 0
- ✓ which PHY is attached to slave 1
- ✓ what MAC addresses to use
- ✓ what link mode each port uses
- ✓ any VLAN separation
- ✓ any fixed-link mode

This is ESSENTIAL information to bring up eth0 and eth1.

If you want, I can explain *next*:

- ✓ How the driver registers each slave as a net_device
- ✓ How cswp_slaves are mapped to hardware ports
- ✓ How link status is monitored via MDIO
- ✓ How cswp_ndo_open() uses this information
- ✓ How interrupts/NAPI relate to each slave

Just say **NEXT**.

soc_device_match function use in probe function

Soc matching nad after it

Friday, December 5, 2025 5:24 PM

Let's break this down **line-by-line in the simplest possible way**, and explain what **specific action** each line performs in CPSW Ethernet driver initialization.

1. `soc = soc_device_match(cpsw_soc_devices);`

 What this does?

- The driver checks what **SoC (System-on-chip)** your kernel is running on.
- `cpsw_soc_devices` is a table of supported SoC models, example:
 - AM335x
 - AM437x
 - AM571x
 - AM572x
 - J721e
 - etc.

 If the running SoC matches one entry, the function returns the matching struct `soc_device_attribute`.

 Why does driver do this?

Different TI SoCs have different quirks/bugs (hardware differences).

Example:

- Some SoCs have **broken CPSW interrupt logic**
- Some need **workarounds for RX/TX ordering**
- Some need **special clock handling**
- Some generate **spurious IRQs**

So the driver checks:

 "Which SoC am I running on?"

2. `if (soc) cpsw->quirk_irq = true;`

 What this does?

If SoC model matches a known SoC with interrupt bug →

Set:

```
cpsw->quirk_irq = true;
```

 Meaning?

This tells the driver:

"This SoC uses a special IRQ handling workaround".

Later in the driver, you will see behavior like:

```
if (cpsw->quirk_irq)
    handle_irq_in_special_way();
else
```

```
handle_irq_normally();
```

⌚ Why needed?

Some TI SoCs:

- Generate IRQ before writing the interrupt status register
- Require read-before-write ordering
- Need extra register flush
- Use combined IRQs instead of per-channel IRQs

The quirk flag enables the right workaround.

✓ 3. data = &cpsw->data;

🔍 What this does?

cpsw->data already contains CPSW information read from **Device Tree**:

- Number of slaves (ports)
- PHY info (phy-handle / phy-id)
- MAC addresses
- RGMII/MII mode
- VLAN use
- Ale info
- Switch type

So this line just stores a pointer to that.

✓ 4. Allocate array of slave ports

```
cpsw->slaves = devm_kcalloc(dev,  
                           data->slaves,  
                           sizeof(struct cpsw_slave),  
                           GFP_KERNEL);
```

🔍 What this does?

This allocates memory for **all CPSW slave ports**.

Example:

If CPSW has 2 ports:

```
data->slaves = 2
```

Then:

```
cpsw->slaves = array of 2 elements
```

[0] → CPSW Slave Port 0

[1] → CPSW Slave Port 1

Each struct cpsw_slave represents:

- port registers
- statistics
- link state
- comparison to PHY device
- VLAN membership

and much more.

Why use devm_kalloc()?

devm_ → device-managed memory

Meaning:

- Automatically freed when driver is removed
- No manual kfree() needed

What specific role these slaves play?

Each slave corresponds to one **external Ethernet port** on the chip.

Example for AM335x:

CPSW Slave ID	External Port
Slave 0	Ethernet Port 1
Slave 1	Ethernet Port 2

Driver sets up:

- DMA channels
- RX flow mapping
- ALE port configuration
- MII/RGMII pins
- PHY link monitoring

For each slave using this structure.

5. Allocation failure case

```
if (!cpsw->slaves) {
    ret = -ENOMEM;
    goto clean_dt_ret;
}
```

If memory allocation fails → return error.

Final Summary: What SPECIFIC ACTION these lines perform

Line	Action
soc_device_match()	Detect which TI SoC is running
quirk_irq = true	Enable special IRQ fix for certain SoCs

data = &cpsw->data	Point to CPSW config read from device tree
devm_kcalloc()	Allocate memory for all slave ports
Error check	Handle memory failure

If you want next:

I can explain:

- ✓ struct cpsw_slave (each field explained)
- ✓ How IRQ quirk changes actual interrupt behavior
- ✓ CPSW slave port RX/TX pipeline
- ✓ Difference between struct cpsw_common and struct cpsw_priv
- ✓ What happens after this allocation (full init sequence)

Just say "**explain cpsw_slave structure**" or anything you want next.

Cpsw hanler

Monday, December 8, 2025 10:55 AM

Let's explain these lines **exactly in the context of the CPSW Ethernet driver**, including **DMA channel numbers**, **why quirk IRQ changes it**, **what TX/RX handlers do**, and **why cpsw_split_res()** is called next.

This part of the CPSW driver is **CPDMA initialization** — the DMA engine that moves packets to/from memory.

1. ch = cpsw->quirk_irq ? 0 : 7;

🔍 What does this do?

Some TI SoCs require using **DMA channel 0** for TX

Some require using **DMA channel 7** for TX

This is because:

- Older silicon used **CPDMA channel 7** for TX
 - Newer silicon or particular SoCs (Sitara variants) use **channel 0** (quirk)
- quirk_irq means “use channel 0 because this SoC behaves differently”.

Example:

SoC	TX DMA channel
AM335x	7
AM437x	0 (quirk)
DRA7	7

So the line switches based on detected SoC.

2. Creating TX DMA channel

cpsw->txv[0].ch = cpdma_chan_create(cpsw->dma, ch, cpsw_tx_handler, 0);

What is happening?

We tell the CPDMA engine:

“Create a TX DMA channel using hardware channel ch.

When a TX completes, call cpsw_tx_handler.”

🔍 What does CPDMA channel do?

TX CPDMA channel:

- Takes packet buffers prepared by driver
- Pushes them to CPSW GMAC
- Updates descriptors
- Raises interrupt when done (unless NAPI polling mode)

🔍 Arguments meaning:

Argument	Meaning
cpsw->dma	CPDMA engine object initialized earlier
ch	TX channel index (0 or 7)
cpsw_tx_handler	Function to call when TX completes
0	Channel parameter (TX=0; RX=1 normally)

What does TX handler do?

cpsw_tx_handler() frees completed TX descriptors:

- Marks descriptors free for reuse
- Unmaps DMA buffers
- Notifies netdev TX queue that space is available

TX handler is ALWAYS FAST.

If TX DMA channel creation fails:

```
goto clean_cpts;
```

We must abort probe because driver cannot function without TX DMA.

3. Creating RX DMA channel

```
cpsw->rxv[0].ch = cpdma_chan_create(cpsw->dma, 0, cpsw_rx_handler, 1);
```

RX channel **ALWAYS** uses CPDMA channel 0

This is how CPDMA hardware is designed.

RX channel responsibilities:

i RX CPDMA channel:

- Monitors RX descriptor ring
- Writes incoming packets into DMA buffers
- Marks descriptor as full when packet received
- Raises RX interrupt

Arguments:

Field	Meaning
cpsw->dma	CPDMA engine
0	Hardware RX channel index
cpsw_rx_handler	RX completion callback
1	RX channel flag

RX handler job:

cpsw_rx_handler() is called when a packet arrives.

It:

- Retrieves packet from descriptor
- Allocates a new buffer for descriptor
- Pushes packet to NAPI poll function
- Wakes NAPI to process RX packets

This is the **first stage of RX path**.

If RX channel fails:

Driver cannot receive packets → abort probe.

4. cpsw_split_res(cpsw); — What does this do?

This function sets up **resource splitting between multiple slaves (ports)**.

CPSW has multiple slave ports:

Example:

- Slave 0 → eth0
- Slave 1 → eth1

But RX/TX DMA is **shared**.

cpsw_split_res() divides the resources:

- Creates separate NAPI contexts for each slave
- Splits TX rings logically between ports
- Assigns VLAN reservation if dual-emac mode
- Configures per-port DMA view
- Sets per-slave max frame sizes
- Prepares CPDMA descriptors for both ports

❖ Why do we need this?

CPSW is a *switch*, meaning:

- Port 0 and port 1 share DMA
 - But Linux wants each port to behave like a **separate NIC**
- cpsw_split_res() ensures:
- TX resources are partitioned
 - RX packets are routed to correct netdev
 - NAPI instance knows which port is responsible
 - Dual EMAC mode works properly

⌚ PUTTING IT ALL TOGETHER

What these few lines achieve:

1. Determine correct TX DMA channel based on SoC quirks
2. Create TX DMA channel for sending packets
3. Create RX DMA channel for receiving packets
4. Split DMA/NAPI resources between slave ports

Without this step:

- CPSW cannot send packets
- CPSW cannot receive packets
- Multiple network interfaces cannot function

This is core CPDMA initialization.

🚀 If you want next:

I can explain the following sections in detail:

- ✓ cpsw_tx_handler() internal flow
- ✓ cpsw_rx_handler() internal flow
- ✓ How CPDMA rings and descriptors work
- ✓ How packets move from device → DMA → NAPI → kernel
- ✓ How CPSW supports multiple netdevs

Just tell me which one you want next!

Let's break this down very clearly and deeply so you understand **every structure, field, and logic flow** that appears in this block of CPSW initialization code.

This part is the **network-device setup** portion of the TI CPSW (TI Ethernet Switch) driver.

◇ 1. `ndev = devm_alloc_etherdev_mqs(...)`

What it does:

This allocates a **struct net_device** with:

- Memory managed by devm (auto-freed on driver detach)
- Extra private memory for driver = sizeof(struct cpsw_priv)
- Multi-queue support: CPSW_MAX_QUEUES TX and RX queues.

Function prototype:

```
struct net_device *devm_alloc_etherdev_mqs(struct device *dev,  
                                         int sizeof_priv,  
                                         unsigned int txqs,  
                                         unsigned int rxqs);
```

Result:

- You get a net_device structure
- Followed by private memory for your driver (cpsw_priv)

◇ 2. priv = netdev_priv(ndev);

The kernel stores the driver-private structure immediately after struct net_device.

Here:

```
struct cpsw_priv *priv;  
priv = netdev_priv(ndev);
```

This gives you a pointer to **cpsw_priv**, the main driver state for an individual Ethernet interface.

◇ 3. priv->cpsw = cpsw;

CPSW driver splits data into two layers:

Global SoC Switch State → struct cpsw_common *cpsw

This contains:

- ALE (Address Lookup Engine)
- DMA engine
- Slaves (ports)
- IRQs
- Memory regions
- MDIO
- Hardware version, quirks, etc.

Per-netdev State → struct cpsw_priv

Used by Linux networking stack for one Ethernet interface.

So:

```
priv->cpsw = cpsw;
```

links the per-netdev instance to the global switch context.

◇ 4. priv->ndev = ndev; and priv->dev = dev;

Saved for convenience:

- priv->ndev → the Linux net_device
- priv->dev → struct device (platform device dev)

These are needed for logging, DMA mappings, etc.

◇ 5. priv->msg_enable = netif_msg_init(debug_level, **CPSW_DEBUG**);

This sets which debug messages will be printed.

debug_level comes from module parameter or DT.

CPSW_DEBUG is a bitmask defining the meaning of each bit:

- link state
- TX completion
- RX handling
- MDIO activity
- etc.

netif_msg_init() maps debug level to bitmask.

◇ 6. `priv->emac_port = 0;`

This selects which CPSW slave port this netdev represents.

CPSW can expose:

- 1 Ethernet port (most TI SoCs)
- 2 Ethernet ports (like AM335x)
- 1 internal host port

Usually port 0 is EMAC (external MAC).

◇ 7. Setting MAC Address

```
if (is_valid_ether_addr(data->slave_data[0].mac_addr)) {  
    memcpy(priv->mac_addr, data->slave_data[0].mac_addr, ETH_ALEN);  
    dev_info(dev, "Detected MACID = %pM\n", priv->mac_addr);  
} else {  
    eth_random_addr(priv->mac_addr);  
    dev_info(dev, "Random MACID = %pM\n", priv->mac_addr);  
}
```

◇ Where does `data->slave_data[0].mac_addr` come from?

From Device Tree parsing earlier.

DT example:

```
slave@0 {  
    mac-address = [00 14 22 AA BB CC];  
};
```

Driver loads it into:

```
data->slave_data[i].mac_addr[6];
```

◇ 8. What is `struct cpsw_priv`? (Very important)

This is the per-netdev private state.

Common fields:

```
struct cpsw_priv {  
    struct cpsw_common *cpsw; // Global switch state  
    struct net_device *ndev;  
    struct device *dev;  
  
    u8 mac_addr[ETH_ALEN]; // MAC address of this port  
    int emac_port; // Port number  
    u32 msg_enable; // Debugging flags  
  
    struct napi_struct napi_rx; // RX polling context  
    struct napi_struct napi_tx; // TX polling context  
  
    spinlock_t lock; // Serialize TX completion, etc  
    ...  
};
```

This structure ties together:

- CPSW DMA channels
- NAPI contexts
- Statistics
- Per-port settings
- MAC address
- Link state

◇ 9. Why random MAC sometimes?

If DT does NOT provide a MAC, Linux generates one so interface can still come up and communicate locally.

Common for:

- Embedded boards without EEPROM MAC
- Development boards like BBB or AM335x EVM

◇ 10. Flow Summary

Full pipeline:

- ✓ Allocate net_device → devm_alloc_etherdev_mqs
- ✓ Get driver private state → netdev_priv
- ✓ Attach global CPSW context → priv->cpsw
- ✓ Store pointers to dev + ndev
- ✓ Enable debug flags
- ✓ Assign port number
- ✓ Load MAC address from DT
- ✓ Else generate a random MAC

After this, next steps will be:

- dev_set_mac_address()
- eth_hw_addr_set()
- register_netdev(ndev)
- Create NAPI contexts
- Enable DMA channels

If you want:

I can also explain:

- ✓ struct cpsw_common in full
- ✓ struct cpsw_slave in detail
- ✓ How CPSW DMA rings work
- ✓ Full CPSW TX and RX flow
- ✓ How the host port and ALE entries map packets
- ✓ How interrupt coalescing works

Just tell me **what part you want next**.

Let's break this block **line-by-line**, with clear explanations of what each field, structure, and function really does inside the network subsystem and CPSW driver.

This part completes the **net_device setup**, connects it to CPSW, and binds NAPI.

1. Set the MAC address into net_device

```
memcpy(ndev->dev_addr, priv->mac_addr, ETH_ALEN);
```

What happens here:

- Earlier, we stored MAC into priv->mac_addr.
- Now we copy that into the official struct net_device MAC address field:
ndev->dev_addr[6]

This is the MAC Linux exposes to userspace, and what the NIC uses for TX.

Internally:

eth_hw_addr_set(ndev, mac); is usually used, but direct memcpy is also common.

2. Bind this netdev to CPSW slave port

```
cpsw->slaves[0].ndev = ndev;
```

Meaning:

Each CPSW slave port (port 1, port 2) corresponds to a PHY/MAC interface.

struct cpsw_slave cpsw->slaves[] contains:

- port number
- PHY connection
- register base
- slave configuration
- attached net_device

So this line registers that **slave port 0** (usually "port 1" externally) is associated with this network interface.

Later TX/RX logic uses this information to:

- pick DMA channels
- configure ALE port settings
- program MAC control registers

3. Enable hardware VLAN filtering

```
ndev->features |= NETIF_F_HW_VLAN_CTAG_FILTER | NETIF_F_HW_VLAN_CTAG_RX;
```

Meaning:

- NETIF_F_HW_VLAN_CTAG_FILTER
Hardware can filter VLAN packets based on VLAN IDs in the ALE.
- NETIF_F_HW_VLAN_CTAG_RX
Hardware can tag VLANs on RX without the CPU parsing frames manually.

Benefits:

- Fast VLAN switching
- CPU load reduced
- ALE can drop non-matching VLANs

CPSW ALE supports VLAN-aware forwarding, hence driver enables these flags.

4. Assign network operations (net_device_ops)

```
ndev->netdev_ops = &cpsw_netdev_ops;
```

This struct contains all kernel callbacks for the device:

Examples:

```
int (*ndo_open)(struct net_device *ndev);
int (*ndo_stop)(struct net_device *ndev);
netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb, struct net_device *ndev);
int (*ndo_set_mac_address)(...);
```

```
int (*ndo_change_mtu)(...);
```

These functions implement:

- Interface up/down (ifconfig up)
- Transmitting packets (ndo_start_xmit)
- Setting MAC address
- Changing MTU
- Managing TX queue
- Statistics

This is the **heart of every network driver.**

5. Assign ethtool operations

```
ndev->ethtool_ops = &cpsw_ethtool_ops;
```

What it does:

This connects ethtool support for:

- Getting link settings
- Forcing speed/duplex
- Reading statistics
- Getting/setting ring sizes
- Dumping registers, EEPROM
- Getting pause frame settings

Example commands:

```
ethtool eth0  
ethtool -a eth0  
ethtool -S eth0
```

All of that is implemented through struct ethtool_ops.

6. Add NAPI for RX

```
netif_napi_add(ndev, &cpsw->napi_rx,  
    cpsw->quirk_irq ? cpsw_rx_poll : cpsw_rx_mq_poll,  
    CPSW_POLL_WEIGHT);
```

Meaning:

- Creates a NAPI polling context for RX
- Binds to the main net_device
- Chooses poll handler based on hardware quirk

Two possible handlers:

- cpsw_rx_poll — legacy single-queue RX handler (for quirk SoCs)
- cpsw_rx_mq_poll — multi-queue handler for newer CPSW

Why NAPI:

- Reduces interrupt load
- Batches RX packets
- Improves throughput
- Prevents RX storm from overwhelming CPU

POLL_WEIGHT:

Defines how many packets per NAPI poll run:

CPSW_POLL_WEIGHT (typically 64 or 128)

7. Add NAPI for TX

```
netif_tx_napi_add(ndev, &cpsw->napi_tx,
```

```
cpsw->quirk_irq ? cpsw_tx_poll : cpsw_tx_mq_poll,  
CPSW_POLL_WEIGHT);
```

Why TX NAPI?

CPSW uses TX completion interrupts.

When a TX completion happens:

- You've to clean descriptors
- Free SKBs
- Restart queues if stopped

Using NAPI:

- Defers TX cleanup to poll context
- Avoids burst interrupt storms
- Consolidates multiple completions into one poll

TX poll handler:

- cpsw_tx_poll for old hardware
- cpsw_tx_mq_poll for multi-queue SoCs

◇ Summary of What This Block Completes

This section finalizes the network interface by:

- ✓ Setting MAC address
- ✓ Linking netdev → CPSW hardware port
- ✓ Enabling hardware VLAN handling
- ✓ Attaching netdev operations
- ✓ Attaching ethtool operations
- ✓ Enabling NAPI based RX processing
- ✓ Enabling NAPI based TX completion handling

After this, the driver will call:

`register_netdev(ndev)`

And the interface becomes visible as:

eth0

💧 If you want, I can continue explaining:

- What happens in `npo_open()`
- How TX DMA works (`cpdma_chan`)
- Full TX flow: SKB → Descriptor → DMA → CPSW port
- Full RX flow: DMA → SKB → Network stack
- ALE programming (port states, VLANs, MAC entries)
- Interrupt handling
- Switch host port logic

Just tell me **which part you want next**.

Let's go through this block **slowly and deeply**, because this is the point where your CPSW network interface finally becomes a formal Linux network device (eth0, eth1, etc.).

★ 1. Register the network device with the Linux networking core

```
SET_NETDEV_DEV(ndev, dev);
```

! What this does:

Links the `net_device` (`ndev`) to the parent `struct device` (`dev`) in the driver model.

This allows:

- sysfs to show a proper hierarchy (net device under platform device)
- Proper reference counting
- Runtime PM integration
- Udev rules to work correctly

In sysfs, it ensures:

```
/sys/class/net/eth0/device → /sys/devices/platform/cpsw
```

★ 2. Connect the DT node to the net_device

```
ndev->dev.of_node = csw->slaves[0].data->slave_node;
```

Why this matters:

- This tells the kernel: "This network device corresponds to this DT node".
- Allows network stack and ethtool to read DT properties of the slave port.
- Allows phy and phylink to bind correctly using the DT.

Basically, this line ensures that the eth0 interface knows **which slave port in device tree it belongs to**.

★ 3. Now register the network device

```
ret = register_netdev(ndev);
```

💡 This is the MOST important moment.

After this call:

- eth0 is created.
- It appears in ip link, ifconfig, /sys/class/net/.
- User can bring it up with:
ip link set eth0 up
- Kernel assigns an ifindex.
- Network stack internal structures are initialized.

Internally, register_netdev() does:

1. Allocates netdev index
2. Allocates qdisc (default pfifo_fast or mq)
3. Calls ndo_init (if provided)
4. Registers with rtnetlink
5. Triggers udev events
6. Makes interface visible to userspace

So at this point, your CPSW driver becomes a usable network interface.

If registration fails, the driver will abort probe.

★ 4. Check and handle dual-emac mode

```
if (cpsz->data.dual_emac) {  
    ret = cpsz_probe_dual_emac(priv);
```

CPSW can operate in two modes:

◇ Mode 1: Switch Mode (Default)

CPSW has:

- 1 Host port (connected to CPU)
- 2 External slave ports (connected to PHYs)

In this mode:

- Hardware works like a 3-port switch
- Linux sees **ONE** network interface (eth0)
- ALE decides forwarding

◇ Mode 2: Dual EMAC Mode

Each slave port becomes its own network interface:

eth0 → Slave 1

eth1 → Slave 2

The switch is turned into two independent MAC controllers sharing host port.

Why dual-emac exists:

- Industrial devices needing separate independent networks
- Zero cross-traffic
- Each port has its own:
 - PHY
 - MAC address
 - DMA channels
 - VLAN separation

★ What does `cpsw_probe_dual_emac()` do?

It performs **the entire netdev creation process again for the second slave**:

- Allocate a 2nd `net_device`
- Set MAC address from `slave_data[1]`
- Attach `ethtool_ops`
- Setup NAPI for port 2
- Register as `eth1`

Also configures ALE to isolate slaves using VLANs:

slave1: VLAN ID 1

slave2: VLAN ID 2

So packets never leak from one port to other.

If `dual_emac` setup fails, driver unregisters primary netdev and aborts.

★ Final Peeking at Control Flow

Putting it together:

► Primary EMAC (slave 0)

- Create `ndev`
- `init ops + ethtool`
- `setup NAPI`
- `assign MAC`
- `register as eth0`

► If `dual_emac` is enabled

- `create 2nd netdev`
- `setup second port`
- `configure VLANs to isolate`

- register as eth1

So your system ends up with:

eth0 → slave port 1

eth1 → slave port 2 (dual-emac only)

⌚ If you want, next I can explain:

- ✓ What happens during ifconfig eth0 up / ip link set eth0 up (ndo_open)
- ✓ TX flow after registration
- ✓ RX flow after registration
- ✓ How CPSW allocates DMA descriptors after registration
- ✓ ALE initialization after netdev registration

Just tell me what you want next.

Alright — now let's break this down **very clearly** and in the context of CPSW hardware and NAPI behavior.

This block is about **registering interrupts** for TX, RX, and MISC events.

★ FIRST — Important Point

CPSW runs **NAPI-driven RX/TX processing**, but **interrupts are still required to notify NAPI to start polling**.

NAPI works like this:

7. Interrupt fires → NAPI poll scheduled
8. Polling runs → driver empties all packets
9. When queue empty → re-enable interrupts

! So interrupts are not used for per-packet processing, but for waking up NAPI.

★ The Code You Asked About

◇ 1. Request RX interrupt

```
ret = devm_request_irq(dev, cpsw->irqs_table[0],
                      cpsw_rx_interrupt,
                      0, dev_name(dev), cpsw);
```

✓ What this interrupt does:

- This fires when **RX DMA channel posts a completion interrupt**
 - Meaning **at least one frame has been DMA'd into RX descriptors**
- ✓ What `cpsw_rx_interrupt()` does:
- **disables RX IRQ**
 - schedules NAPI poll:
`napi_schedule(&cpsw->napi_rx);`
 - returns `IRQ_HANDLED`

Then NAPI busy loops (polling mode) to process multiple packets.

◇ 2. Request TX interrupt

```
ret = devm_request_irq(dev, csw->irqs_table[1],  
                      csw_tx_interrupt,  
                      0, dev_name(&pdev->dev), csw);
```

✓ What TX interrupt does:

This interrupt fires when:

- DMA engine has transmitted (completed) a TX packet
 - TX descriptor gets freed
- ✓ Why TX interrupt is needed:
- To wake NAPI_TX polling
 - To free skb's
 - To wake the network queue if it was stopped (`netif_stop_queue`)

Example:

10. Driver sends a packet → descriptor in TX queue becomes “owned by DMA”
11. Hardware sends it out
12. TX interrupt fires
13. `csw_tx_interrupt` schedules NAPI TX poll
14. NAPI frees descriptors and cleans queues

◇ 3. Request MISC interrupt (optional)

```
if (!csw->cpts)
```

```
    goto skip_cpts;
```

```
ret = devm_request_irq(&pdev->dev, csw->misc_irq,  
                      csw_misc_interrupt,
```

```
0, dev_name(&pdev->dev), cpts);
```

✓ This interrupt is for:

- **CPSW CPTS: Common Platform Time Sync**
- Used for PTP (Precision Time Protocol) and timestamps

CPTS generates events:

- Timestamp events
- PTP clock sync events
- Overflow events

This interrupt notifies software:

“Hey, CPSW timestamp engine has something in FIFO.”

CPTS is used for:

- hardware timestamping for IEEE 1588 PTP
- high precision timing in industrial systems

◇ 4. Enable CPTS IRQ polling

```
cpts_set_irqpoll(cpsw->cpts, false);
```

This configures how CPTS manages timestamp events:

- true = use polling mode
- false = use interrupts

So here they enable **interrupt mode** for CPTS.

★ Why RX_THRESHOLD and MISC IRQs Are Disabled in This Driver

From the comment:

```
/* Note: RX_THRESHOLD and MISC IRQs are always kept disabled */
```

RX_THRESHOLD interrupt

This fires **when RX queue reaches a certain fill level**

Not needed because NAPI poll handles draining RX buffers.

RX interrupt (the one used) means “new packet arrived”

RX_THRESHOLD interrupt means “too many packets queued”

CPSW driver uses NAPI so doesn't need a threshold interrupt — poller will empty the queue.

★ Summary in Driver Context

Interrupt	Why needed?	What handler does?
-----------	-------------	--------------------

RX	New packet DMA completed	Disable IRQ → schedule NAPI RX
TX	Packet transmitted, descriptor freed	Disable IRQ → schedule NAPI TX
MISC (CPTS)	Timestamp/clock events	Read timestamp FIFO
RX_THRESHOLD	UNUSED	NAPI replaces this need

Other MISC UNUSED Not needed in this driver

★ Example Scenario (Very Important)

Imagine a packet arrives on the wire:

15. RX DMA moves it into RX descriptor ring
16. Hardware raises **RX IRQ**
17. `cpsw_rx_interrupt()` runs
 - disables RX IRQ
 - schedules NAPI poll
18. NAPI poll begins (busy loop):
 - read 32 packets from DMA
 - refill descriptors
 - continue until queue empty
19. Poll completes
20. Driver re-enables RX IRQ

So interrupts only happen when nose of RX queue changes from empty → non-empty.

★ If you want, next I can explain:

- ✓ EXACT flow inside `cpsw_rx_interrupt()`
- ✓ EXACT NAPI RX poll flow (`cpsw_rx_poll()`)
- ✓ How TX interrupt + TX poll cleans descriptors
- ✓ How CPTS timestamping works internally
- ✓ How DMA IRQs are generated inside CPDMA hardware

Tell me what you want next.

Simple probe code

Monday, December 8, 2025 11:30 AM

```
static int my_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    struct my_priv *priv;
    struct net_device *ndev;
    struct resource *res;
    int ret;

    /* Step 1: Allocate priv + netdev */
    ndev = devm_alloc_etherdev(dev, sizeof(struct my_priv));
    if (!ndev)
        return -ENOMEM;

    priv = netdev_priv(ndev);
    priv->ndev = ndev;
    SET_NETDEV_DEV(ndev, dev);

    /* Step 2: Map registers */
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    priv->regs = devm_ioremap_resource(dev, res);
    if (IS_ERR(priv->regs))
        return PTR_ERR(priv->regs);

    /* Step 3: Get IRQ */
    priv->irq = platform_get_irq(pdev, 0);
    if (priv->irq < 0)
        return priv->irq;

    /* Step 4: Setup NAPI */
    netif_napi_add(ndev, &priv->napi, my_napi_poll, 64);

    /* Step 5: Register IRQ */
    ret = devm_request_irq(dev, priv->irq, my_irq_handler, 0,
                           dev_name(dev), priv);
```

```
if (ret)
    return ret;

/* Step 6: Setup MAC address */
eth_random_addr(ndev->dev_addr);

/* Step 7: Register netdev */
ret = register_netdev(ndev);
if (ret)
    return ret;

dev_info(dev, "my-eth driver probed\n");
return 0;
}
```

Cpsw_ndo_open

Monday, December 8, 2025 11:45 AM

Below is a **deep, clear, beginner-friendly explanation** of the next section of `cpsw_ndo_open()` — the part where the driver declares the number of TX/RX queues and prints CPSW hardware version.

This section is essential because **CPSW supports multiple hardware queues**, and the kernel networking layer must know exactly how many to use.



1. Tell the networking stack how many TX queues exist

```
ret = netif_set_real_num_tx_queues(ndev, cpsw->tx_ch_num);
if (ret) {
    dev_err(priv->dev, "cannot set real number of tx queues\n");
    goto err_cleanup;
}
```



Why do we need this?

When the driver creates the `net_device` during probe, it allocates:

```
devm_alloc_etherdev_mqs(dev, sizeof(struct cpsw_priv),
                        CPSW_MAX_QUEUES, CPSW_MAX_QUEUES);
```

This means **the maximum number** of queues are allocated (e.g., 8 TX + 8 RX).

BUT the actual hardware might be configured for:

- Only 1 TX + 1 RX
- Or 4 TX
- Or 2 RX
- Depending on SoC capabilities
- Depending on DTS properties

So before the interface is brought up, the driver must inform the kernel:

“Out of the max queues allocated, we will use exactly N TX queues.”

This is what `netif_set_real_num_tx_queues()` does.



What happens inside?

The kernel updates:

```
ndev->num_tx_queues = cpsw->tx_ch_num;
```

So:

- Qdisc scheduling works only for valid queues
- NAPI binds correctly
- CPU queue selection works
- TX hashing (RPS/XPS) maps correctly

Failing this would cause queue mismatch → packet drops, crashes, or unbalanced load.



2. Tell the networking stack how many RX queues exist

```
ret = netif_set_real_num_rx_queues(ndev, cpsw->rx_ch_num);

if (ret) {
    dev_err(priv->dev, "cannot set real number of rx queues\n");
    goto err_cleanup;
}
```

Same logic, but for RX.

Why RX matters?

RX queue count affects:

- NAPI instance count
- IRQ distribution
- DMA channel mapping
- RPS (Receive Packet Steering)
- How the kernel wakes NAPI poll functions

CPSW typically uses:

- cpsw->rx_ch_num = 1 (classic CPSW2G)
- But CPSW3G / CPSW1G in AM335x/AM437x may support more in the future

The driver must correctly tell the stack before starting RX DMA.



3. Read CPSW hardware version

```
reg = cpsw->version;
```

During probe(), CPSW earlier read the version register:

CPSW_SS_VERSION

This contains:

- Major version (e.g., 1, 2, 3)
- Minor version
- RTL patch version



4. Print hardware version

```
dev_info(priv->dev, "initializing cpsw version %d.%d (%d)\n",
         CPSW_MAJOR_VERSION(reg), CPSW_MINOR_VERSION(reg),
         CPSW_RTL_VERSION(reg));
```

What this prints?

Example:

initializing cpsw version 2.1 (3)

Meaning:

- **CPSW major version = 2** → CPSW2G switch
- **minor version = 1**
- **RTL revision = 3** → silicon-level patch or IP version
[Why is this important?](#)

Different CPSW versions behave differently:

Version	SoC / Meaning
1.x	Older DaVinci chips
2.x	AM335x, AM437x (common)
3.x	AM57xx, AM65xx
5.x	CPSW5G on newer Sitara

The driver sometimes applies:

- Bug workarounds
- DMA quirks
- ALE behavior changes

based on this version.

So printing it helps debug and confirms the driver knows the right hardware.



SUMMARY: What did this code block do?

Step	Purpose
netif_set_real_num_tx_queues()	Tell kernel actual TX channels CPSW supports
netif_set_real_num_rx_queues()	Tell kernel real RX channels
Version extraction & print	Hardware identification, debugging, choosing quirks

This is **queue negotiation + hardware identification** before full interface bring-up.



What comes next in ndo_open()?

After this block, CPSW ndo_open usually continues with:

1. Initialize ALE (Address Lookup Engine)
2. Configure host port
3. Reset & start CPDMA engine
4. Submit RX buffers to DMA descriptors
5. Enable TX/RX interrupts
6. Start NAPI
7. Connect + start PHY
8. Enable slave ports

If you want, I can continue explaining the NEXT block from the CPSW open function.

Just paste it or say:

👉 “Explain the next part”

Great — now let's decode this next part of `cpsw_ndo_open()` in a slow, detailed, hardware-oriented manner.

This is where CPSW begins actually configuring the switch fabric (host port + slave ports), and prepares VLAN configuration before traffic begins to flow.

✓ 1. usage_count check

```
if (!cpsw->usage_count)  
    cpsw_init_host_port(priv);
```

🔍 What is `usage_count`?

- CPSW is a **switch with multiple ports**.
- More than one `net_device` may use the CPSW common block (e.g., in dual EMAC mode).
- `usage_count` tracks **how many interfaces** (e.g., `eth0, eth1`) have opened CPSW.

Why do we check it?

If this is the **first** time any interface is opened:

- Host port must be completely initialized.
- ALE tables must be reset.
- CPDMA channels must be reset.
- Statistics counters must be configured.

If it's already opened (second/third interface):

- Host port is already running → no need to reset.

🔧 What does `cpsw_init_host_port()` do?

This function configures:

- Host port VLAN behavior
- Host port ingress/egress settings
- Priority mapping
- RX/Tx flow control
- ALE port control bits
- Disables default host port VID if dual EMAC
- Sets up statistics collection

Essentially:

It prepares the internal CPU-facing port so Linux can transmit and receive frames through the switch.

✓ 2. Initialize each slave port

```
for_each_slave(priv, cpsw_slave_open, priv);
```

What is a *slave* here?

Each “slave” is a **MAC port** connected to:

- The PHY via MII/RMII/RGMII

- The switch fabric
- A dedicated DMA flow

Example:

Slave 0 → eth0 → PHY0

Slave 1 → eth1 → PHY1

What does `cpsw_slave_open()` do?

For each port, it:

1. **Configures the MAC port registers**
 - MAC control
 - GMII_SEL interface type
 - Duplex / speed settings
 - RX/TX enable bits
2. **Sets the source MAC address for that port**
3. **Connects the port to the ALE**
 - Assigns port number inside the switch
 - Enables forwarding
4. **Starts the PHY**
 - `mdio_register`
 - `phy_start`
 - link state callbacks
5. **Configures port VLAN membership**

So effectively:

It physically activates the external Ethernet port and binds it with its PHY.

⌚ After this step, all physical ports are ready but traffic isn't flowing yet — VLAN setup is next.

3. Add default VLAN

```
if (!cpsw->data.dual_emac)
    cpsw_add_default_vlan(priv);
else
    cpsw_ale_add_vlan(cpsw->ale, cpsw->data.default_vlan,
                       ALE_ALL_PORTS, ALE_ALL_PORTS, 0, 0);
```

This section determines how frames move between ports inside the switch.



Case 1: Normal CPSW mode (NOT dual EMAC)

```
cpsw_add_default_vlan(priv);
```

Meaning:

- All ports (host + slave ports) join **one VLAN**.
- It behaves like a normal Ethernet switch.
- Frames can move from:
 - Slave0 ↔ Slave1
 - Slave ↔ Host port (Linux)

Switching is enabled among all ports.

Perfect for "switch-like" operation.



Case 2: Dual EMAC mode (each slave = separate interface)

`cpsw_ale_add_vlan(... ALE_ALL_PORTS ...);`

But with special handling.

What is dual EMAC?

TI added a mode where:

- CPSW acts like **two independent MACs**, not a switch.
- Slave0 = eth0
- Slave1 = eth1
- They do NOT forward frames between each other.

Internally, CPSW uses VLAN separation to isolate ports.

What does this line do?

```
cpsw_ale_add_vlan(cpsw->ale, cpsw->data.default_vlan,
                    ALE_ALL_PORTS, ALE_ALL_PORTS, 0, 0);
```

It installs a VLAN entry in ALE:

- VLAN ID = `default_vlan`
- Allowed ports = host port + all slave ports
- Untagged = all ports
- No flooding restrictions (0,0)

This step affects how ALE learns and filters MAC addresses.

In dual EMAC, the driver later sets **per-port reserved VLANs** to prevent inter-port traffic.



SUMMARY: What this block of code achieves

Step	Purpose
<code>cpsw_init_host_port()</code>	Setup host port (CPU port) for forwarding
<code>cpsw_slave_open()</code>	Configure each MAC port + PHY + switch routing
Add default VLAN	Set up internal switching / port isolation behavior

This is the "switch fabric bring-up" portion of `ndo_open()`.

In other words:

At this point, the hardware datapath between DMA ⇔ ALE ⇔ MAC ⇔ PHY is being configured.

 If you want, I can now explain the NEXT code block from `cpsw_ndo_open()`.

Just paste it or say:

 “Explain the next part.”

Summary

Thursday, December 11, 2025 2:49 PM

Okay.

Now I will explain **the entire csw_ndo_open() function** in the **SIMPLEST possible manner** — like telling a story of what happens **step-by-step** when the network interface (eth0/eth1) is brought UP.

No advanced kernel words.

No big jargon.

Just clear steps with simple meaning.

★ WHAT IS ndo_open? (very simple)

ndo_open() = Function that runs when you type:

ifconfig eth0 up

or

ip link set eth0 up

This is how the **driver STARTS the hardware**, gets it ready to send/receive packets.

★ Now: SIMPLE STEP-BY-STEP LOGIC OF THE WHOLE FUNCTION

Below is the **entire meaning** of csw_ndo_open() in 14 very simple steps.

✓ STEP 1 – Print info (debug)

Driver prints:

- interface name
- MAC address
- MTU

Just for log/debug.

✓ STEP 2 – Power ON the hardware

pm_runtime_get_sync()

This turns ON CPSW hardware:

- enables clocks
- brings block out of low-power
- makes registers readable
- enables MDIO access

If power-on fails → abort.

STEP 3 – Tell Linux “link is DOWN for now”

`netif_carrier_off(ndev)`

Because:

- We haven't checked PHY yet
- No link established
- cable might not be connected

So Linux should not send packets yet.

STEP 4 – Tell Linux how many TX/RX queues we use

`netif_set_real_num_tx_queues()`
`netif_set_real_num_rx_queues()`

This tell networking core:

- “I have X TX channels”
- “I have Y RX channels”

Without this, Linux does not know how many queues exist.

STEP 5 – Print CPSW version

Reads CPSW version register, prints something like:

`initializing cpsw version 1.3 (18)`

Just informational.

STEP 6 – If first interface being opened → Init HOST PORT

`if (!cpsw->usage_count)`
`cpsw_init_host_port()`

HOST PORT = the connection from Linux stack ↔ CPSW switch.

This sets:

- MAC address
- FIFO behavior
- priority settings
- statistics blocks

Only done once (when first interface opens).

STEP 7 – Open each slave port

for_each_slave(..., csw_slave_open)

For slave0, slave1:

- Configure PHY for that port
- Enable port in ALE
- Configure RGMII/MII mode
- Setup flow control
- Bring MAC port out of reset

This is hardware-specific initialization for each port.

STEP 8 – Add a default VLAN (if switch mode)

If NOT dual-emac mode:

csw_add_default_vlan()

If dual-emac, use reserved VLAN per port.

This configures forwarding rules inside the switch.

STEP 9 – If first interface → setup “shared” hardware

Only do this ONCE (first interface):

disable priority elevation

enable statistics collection

enable flow control

Then:

9A – Enable NAPI

napi_enable(rx)

napi_enable(tx)

NAPI is the polling engine for packet RX/TX.

9B – Enable IRQs (if they were disabled earlier)

If interrupts were off:

enable_irq(RX)

enable_irq(TX)

9C – Create RX queues (for XDP also)

```
cpsw_create_xdp_rxqs()
```

```
cpsw_fill_rx_channels()
```

This sets up the buffers/descriptors for receiving packets.

9D – Enable hardware timestamping if present

```
cpts_register()
```

★ STEP 10 – At this point hardware is READY

All ports initialized

DMA setup completed

NAPI enabled

IRQs enabled

CPSW functional

★ STEP 11 – Increase interface usage count

Driver tracks how many interfaces are up (eth0 only or eth0+eth1).

Used to know when to create/destroy shared resources.

★ STEP 12 – Start PHY

PHY subsystem starts monitoring link:

- link up/down
- speed
- duplex

This will later call:

```
netif_carrier_on()
```

when cable plugged in.

★ STEP 13 – Start TX queue

Linux is allowed to send packets now:

```
netif_start_queue(ndev)
```

★ STEP 14 – DONE

Interface is now **UP**.

SUPER SIMPLE SUMMARY (5 LINES)

When interface goes up:

1. **Power ON** CPSW hardware
2. **Setup DMA** and RX/TX queues
3. **Open each PHY port**
4. **Enable interrupts + NAPIa**
5. **Start TX/RX and PHY monitoring**

That is the entire `cpsw_ndo_open()` in easy language.

If you want, I can also give:

- ✓ Simplified flowcharts
- ✓ Where packets go after `ndo_open`
- ✓ What `ndo_stop()` does after this
- ✓ Full CPSW packet RX path explanation
- ✓ Full CPSW DMA + NAPI walkthrough

Just tell me 😊

CPDMA

Friday, December 12, 2025 3:22 PM

CPDMA — COMPLETE THEORY + INTERNAL WORKING (Very Detailed Notes)

TI CPSW uses a dedicated DMA engine called:

CPDMA = "CP" (control packet) + DMA

It is a lightweight DMA engine used to move Ethernet frames:

- From CPSW hardware → memory (RX path)
- From memory → CPSW hardware (TX path)

It is **not scatter-gather**, not fancy, but extremely efficient.

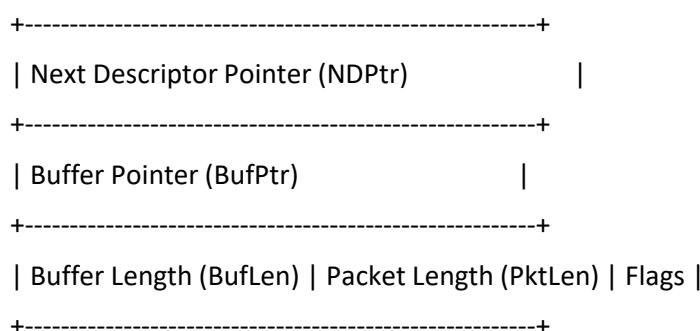
1. CPDMA Basic Architecture

CPDMA contains:

1. **A pool of Buffer Descriptors (BDs)**
 - Each BD describes a packet or packet fragment.
 - Shared between TX and RX.
2. **Channels**
 - RX channels (usually 1)
 - TX channels (can be multiple if multi-queue)
3. Internal **DMA state machine** that:
 - Reads descriptors
 - Moves packet data
 - Updates status fields
 - Writes completion interrupts
4. **Head/Tail pointers**
 - Each channel has **head pointer** in hardware
 - Driver pushes descriptors (tail)
 - Hardware pops descriptors (head)

2. CPDMA Descriptor Format

Each descriptor is a 16-byte structure in hardware-defined RAM:



Simplified fields:

Field	Meaning
next	Pointer to next BD in ring
buffer	Memory address of packet buffer
buf_len	Size of the buffer memory
pkt_len	Actual packet length (RX)
flags	ownership, SOP/EOP, error bits

Important flags:

- **Owner bit** → hardware or software owns BD
- **SOP** → Start of packet
- **EOP** → End of packet
- **Teardown** → Channel teardown marker

3. Descriptor Ownership

CPDMA uses a simple rule:

TX:

- Driver fills descriptor
- Sets ownership to **CPDMA**
- Submits BD to channel
- CPDMA transmits
- Returns BD with ownership to **CPU**

RX:

- Driver allocates empty page and buffer
- Submits BD with ownership = CPDMA
- Hardware receives packet → writes into buffer
- Ownership returned to CPU

4. Descriptor Pool

CPSW driver allocates a big pool:

`cpsw->descs_pool_size`

For example:

- 256 descriptors
- Splits between TX and RX rings
- Each descriptor managed by CPDMA driver

This pool is created inside:

`cpdma_ctlr_create()`

5. CPDMA Channels

Each channel is a queue of descriptors.

RX

Usually only **1 RX channel**:

CPDMA RX channel 0

TX

Multiple TX channels if multi-queue:

tx_channel[0]

tx_channel[1]

...

tx_channel[n]

Inside each channel, CPDMA maintains:

- head pointer (HW consumption)
- tail pointer (SW submission)
- aligned BD ring

6. TX FLOW IN CPDMA (Step-by-Step)

Step 1 — Driver prepares an skb

- skb->data points to packet
- Compute packet length
- Ensure minimum TX size

Step 2 — Driver calls:

cpdma_chan_submit(txch, skb, addr, len, port);

CPDMA driver now:

- Selects next free descriptor
- Writes BD:
 - buffer ptr = skb->data
 - length = packet length
 - SOP/EOP = set
 - owner = CPDMA

Step 3 — Hardware transmits

Hardware fetches BD:

- Reads packet buffer
- Sends frame to CPSW
- Updates stats
- Clears owner bit → CPU owns BD again

Step 4 — TX interrupt handler

Driver calls:

cpdma_chan_process(txch)

It frees TX skb and recycles descriptor.

7. RX FLOW IN CPDMA (Step-by-Step)

Step 1 — Driver allocates empty pages

From page_pool:

```
page = page_pool_alloc_pages();
```

Step 2 — Driver submits RX descriptor

```
bd->buffer = page_address + CPSW_HEADROOM
```

```
bd->buf_len = PAGE_SIZE
```

```
owner = CPDMA
```

Step 3 — Hardware receives packet

When a packet arrives:

- CPDMA writes packet into buffer
- Sets pkt_len
- Clears owner bit → CPU owns BD again
- Fires RX interrupt

Step 4 — RX handler runs

In CPSW driver:

```
cpsw_rx_interrupt → cpdma_chan_process(rxch)
```

The RX BD is read:

- page pointer retrieved
- length parsed
- port determined (dual emac)
- XDP or skb created from buffer

Step 5 — RX descriptor recycled

Driver allocates a **new page** and re-queues descriptor.

8. CPDMA Channel Functions (Driver Side)

✓ `cpdma_ctlr_create()`

Creates DMA controller and BD pool.

✓ `cpdma_chan_create()`

Creates a TX or RX DMA channel.

Called in `probe()`:

```
cpsw->txv[i].ch = cpdma_chan_create(cpsw->dma, i, tx_handler, 0);
```

```
cpsw->rxv[0].ch = cpdma_chan_create(cpsw->dma, 0, rx_handler, 1);
```

✓ `cpdma_chan_submit()`

Submits TX/RX descriptor.

This is the **core function** in TX path.

It:

- Finds a free BD
- Writes BD structure
- Hands BD to hardware
- Kickstarts DMA if idle

✓ cpdma_chan_process()

Called inside IRQ handler.

For RX:

- Return used page
- Extract length
- Detect errors
- Deliver packet to XDP/SKB pipeline

For TX:

- Free completed skbs
- Recycle descriptors

✓ cpdma_ctlr_stop()

Stops all DMA activity (used during ndo_stop)

■ 9. CPDMA Memory Layout

TI CPSW places descriptor RAM at offset:

CPSW2_BD_OFFSET = 0x2000

The driver maps:

BD RAM = csw->regs + CPSW2_BD_OFFSET

Inside this region, CPDMA stores:

- Free descriptor list
- Active RX/TX rings
- Teardown descriptors

■ 10. CPDMA Teardown Mechanism

When shutting down interface:

- Driver writes teardown code to channel pointer
- CPDMA processes teardown BD
- Clears channel state
- Generates teardown completion interrupt

This ensures hardware stops cleanly.

11. How CPDMA interacts with CPSW ports

TX packets include port number:

```
port = priv->emac_port + cpsw->data.dual_emac
```

RX packets include source port encoded in status:

```
CPDMA_RX_SOURCE_PORT(status)
```

So CPDMA:

- Delivers packet per-port
- TX descriptor includes target port
- RX descriptor includes incoming port

12. CPDMA & XDP Interaction

When XDP is enabled:

- RX handler builds an `xdp_buff` from CPDMA page buffer
- XDP runs before skb creation
- XDP_DROP returns → page recycled
- XDP_TX → packet sent back on TX channel
- XDP_REDIRECT → page given to another device
- XDP_PASS → skb continues normal network stack path

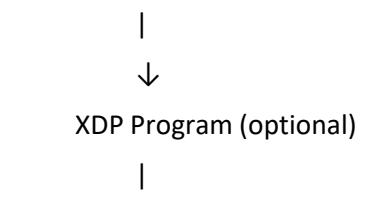
13. Why CPDMA is special (compared to other DMA engines)

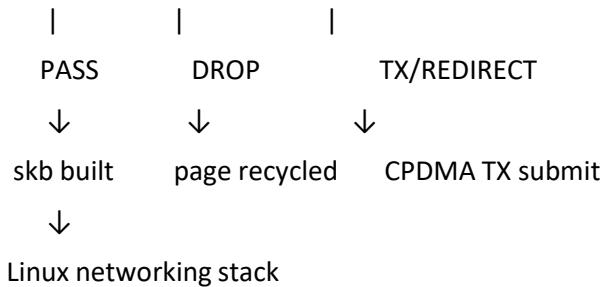
- No scatter-gather
- No huge rings
- Very small hardware state
- Fast descriptor recycling
- Shared BD pool
- Multi-queue supported with ease
- Tight integration with ALE and CPSW switch

This design matches the limitations of embedded AM335x class hardware.

14. Diagram — RX Path Flow

Ethernet MAC → CPDMA → BD → Page Buffer → RX Handler





15. Diagram — TX Path Flow

`skb → ndo_start_xmit →`
`cpdma_chan_submit →`
`Descriptor ring update →`
`Hardware reads → Transmits →`
`BD returned →`
`TX IRQ →`
`Free skb + recycle BD`

Short Summary

CPDMA is a simple yet efficient DMA engine for CPSW Ethernet:

- Uses descriptor rings
- Moves packets via buffer descriptors
- Has TX and RX channels
- Works with XDP and page_pool
- Tie-in with CPSW switch logic via port numbers
- Clean teardown mechanism
- Shared descriptor pool

It is the foundation of:

- RX interrupt handling
- TX queue mapping
- Zero-copy packet delivery
- XDP acceleration

NDO_XMIT

Thursday, December 11, 2025 3:09 PM

Function: `cpsw_ndo_start_xmit()` — Notes (Simple Explanation)

Purpose

This function is the CPSW driver's implementation of the kernel's `.ndo_start_xmit` API.

It is called whenever the Linux networking stack wants the driver to **transmit a packet**.

Incoming path:

socket → TCP/UDP → SKB → `dev_queue_xmit()` → `ndo_start_xmit()`

Initial Variables Explained

```
struct cpsw_priv *priv = netdev_priv(ndev);
```

- Gets private per-interface data of this CPSW port.
- Contains port number, MAC, DMA channel indices, queue mapping.

```
struct cpsw_common *cpsw = priv->cpsw;
```

- Pointer to shared CPSW subsystem data.
- Contains:
 - ALE
 - CPDMA
 - global registers
 - NAPI
 - RX/TX DMA channels
 - IRQ flags

```
struct cpts *cpts = cpsw->cpts;
```

- CPTS = Common Platform Time Sync (for PTP timestamping).
- Used only if timestamping required.

```
struct netdev_queue *txq;
```

- Represents the per-queue TX queue inside the networking stack.
- Used to stop queue if driver runs out of descriptors.

```
struct cpdma_chan *txch;
```

- The CPDMA TX channel used for transmitting this packet.
- CPSW supports multiple TX channels (multi-queue).

```
unsigned int len;
```

- Will store final packet length.

```
int ret, q_idx;
```

- q_idx = selected TX queue index.
- ret = error from CPDMA operations.

```
printk(KERN_INFO "CPSW: Transmitting packet on %s len=%d\n",
      ndev->name, skb->len);
```

- Debug print: logs outgoing packet length and interface name.

Step-wise Logic (so far in the code)

1. **Retrieve per-port private CPSW data (priv).**

Needed to know which CPSW port the packet belongs to.

2. **Retrieve shared CPSW hardware structure (cpsw).**

Needed because TX DMA (CPDMA) and ALE are shared across all ports.

3. **Store pointer to CPTS (cptts).**

Required only when packet needs hardware timestamping.

4. **Define handles for:**

- TX queue (txq)

- TX DMA channel (txch)
 - Packet length (len)
5. **Log debug information about the outgoing SKB.**
Helps verify that packets are reaching the driver.

★ Step 1 — Understanding skb_padto()

✓ What is skb_padto()?

It's a Linux networking helper that:

Pads the tail of an skb with zeros if the skb is shorter than a required minimum length.

Prototype:

```
int skb_padto(struct sk_buff *skb, unsigned int len);
```

✓ Why do we need padding?

Because **Ethernet requires minimum payload length:**

Field	Bytes
Ethernet header	14
Payload	46 bytes minimum
FCS	4 (added by NIC)

So min L2 frame = 60 bytes (or 64 bytes with FCS)

If payload < 46, NIC cannot transmit → must pad.

✓ What skb_padto() really does

6. Checks if skb->len is less than len
7. If yes → allocates extra tailroom
8. Appends zero bytes until length = len
9. Updates skb->len
10. Returns **0 on success, non-zero on error**

✗ When can it fail?

- Memory allocation failure (extremely rare)
- skb is cloned and cannot be modified

★ Step 2 — Why does the driver check if (skb_padto(...))?

```
if (skb_padto(skb, tx_packet_min)) {
    cpsw_err(priv, tx_err, "packet pad failed\n");
```

```
    ndev->stats.tx_dropped++;
    return NET_XMIT_DROP;
}
```

If padding fails:

- ✓ Log an error: csw_err()
- ✓ Increment drop counter: ndev->stats.tx_dropped++
- ✓ Return NET_XMIT_DROP to networking stack

Meaning:

→ **The driver could not ensure minimum Ethernet length → drop the frame.**

★ Step 3 — What is tx_packet_min?

For CPSW on TI SoCs:

```
tx_packet_min = 60 // or 64 depending on configuration
```

This is the **minimum L2 frame size** allowed by hardware (without FCS).

So the driver ensures the outgoing SKB is at least that size.

★ Step 4 — Why is len = skb->len < tx_packet_min ? ... needed?

```
len = skb->len < tx_packet_min ? tx_packet_min : skb->len;
```

After padding:

- If the packet was short → padded → its length is now tx_packet_min
- If it was already long → keep its real length

The variable len will be used later when programming the DMA descriptor:

✓ DMA descriptors need the correct frame size

Otherwise:

- NIC may underrun
- NIC may drop frame

- Incorrect data transferred

So this line simply decides:

“Actual length given to hardware = padded length or real length.”

★ Step-By-Step Flow (Simple)

11. Compute minimum allowed TX packet size (tx_packet_min)
12. Check packet length
13. If too small → call skb_padto() to add zeros
14. If padding fails → drop packet
15. Select final length (padded or real)
16. Pass to DMA and begin transmission

★ Why does Ethernet require padding? (Important theory)

Ethernet uses **CSMA/CD** (in classic half-duplex) and has:

- Minimum frame size = **64 bytes** (incl. FCS)
- NIC adds 4 bytes FCS → driver must supply at least 60 bytes

Shorter frames are called **runt**s → forced to drop.

This design ensures:

- ✓ Receiver detects collisions
- ✓ No ambiguous short frames

Even in full-duplex modern systems, minimum size is preserved.

Full info

Thursday, December 11, 2025 4:14 PM

Below is **clean, crisp, note-style explanation of `cpsw_ndo_start_xmit()` — the transmit (TX) function of the TI CPSW Ethernet driver.**

I will explain **what the function does and why each step exists**, using **simple language + CPSW hardware reasoning**.

What is `cpsw_ndo_start_xmit()`?

It is the **TX (transmit) callback** that the Linux network stack calls whenever a user or kernel code wants to send a packet.

Examples that trigger this function:

- ping
- curl
- TCP/UDP traffic
- Kernel routing
- VLAN transmission

This function **moves the SKB into CPSW hardware DMA (CPDMA)**.

Step-by-step logic (very simplified)

1. Validate and pad packet
2. Handle hardware timestamping (PTP)
3. Select the TX queue / DMA channel
4. Submit packet to CPDMA
5. Manage TX queue stopping/waking

Now detailed step-wise breakdown:

❖ 1. Extract driver-private structures

```
struct cpsw_priv *priv = netdev_priv(ndev);  
struct cpsw_common *cpsw = priv->cpsw;  
struct cpts *cpts = cpsw->cpts;
```

- priv → per-interface private data (eth0 or eth1)
- cpsw → shared CPSW switch/DMA/ALE registers
- cpts → timestamp provider (PTP clock)

Why?

Driver needs common registers + DMA channels + PTP clock.

❖ 2. Ensure packet meets MINIMUM Ethernet size

```
if (skb_padto(skb, tx_packet_min)) { ... }
```

What is happening?

Ethernet minimum frame size = **60 bytes (without CRC)**

If user sends smaller packet (e.g., TCP ACK of 40 bytes),
the driver must **pad** it.

skb_padto() adds padding bytes.

Why?

CPSW hardware **will drop** frames < 60 bytes.

❖ 3. Handle hardware timestamping (PTP)

```
if (skb_shinfo(skb)->tx_flags & SKBTX_HW_TSTAMP &&  
    priv->tx_ts_enabled &&  
    cpts_can_timestamp(cpts, skb))
```

```
skb_shinfo(skb)->tx_flags |= SKBTX_IN_PROGRESS;
```

Why?

Some applications (PTP, IEEE1588) request hardware timestamps.

This block:

- Checks if user requested TX timestamp
- Checks if CPSW supports it for this frame
- Marks SKB as **timestamp in progress**

Later the hardware interrupt will produce timestamp and complete the SKB.

❖ 4. Select correct TX queue / channel

```
q_idx = skb_get_queue_mapping(skb);  
if (q_idx >= csw->tx_ch_num)  
    q_idx = q_idx % csw->tx_ch_num;
```

- The kernel can map different flows to different TX queues (multi-queue support).
- CPSW may have 1, 2, or more hardware TX DMA channels.
If user chooses queue 3 but CPSW has only 2 → wrap it.

Why?

Ensures packet always maps to a valid hardware TX channel.

❖ 5. Pick the DMA channel

```
txch = csw->txv[q_idx].ch;
```

- txv[] = array of TX DMA channels (per queue)
- ch = CPDMA channel struct

❖ 6. Inform kernel timestamping subsystem

```
skb_tx_timestamp(skb);
```

This does **software timestamp** if requested.

Hardware timestamp will come later.

❖ 7. Submit SKB to CPDMA

```
ret = cpdma_chan_submit(txch,  
                        skb,  
                        skb->data,  
                        skb->len,  
                        priv->emac_port + cpsw->data.dual_emac);
```

This is **the most important line**.

CPDMA submit arguments:

Arg	Meaning
txch	Which DMA TX channel
skb	Packet buffer
skb->data	Pointer to actual Ethernet frame
skb->len	Length of packet
priv->emac_port + ...	CPSW switch port number

What CPDMA does:

- Allocates a TX BD (buffer descriptor)
- Fills it with:
 - packet address
 - length
 - port number
- Sets "OWN" bit so CPSW can pick it

- Inserts BD into TX DMA ring
- Kicks CPDMA hardware
If return value $\neq 0 \rightarrow$ submission failed
(likely no free descriptors).

8. Handle descriptor shortage

```
if (!cpdma_check_free_tx_desc(txch)) {
    netif_tx_stop_queue(txq);
    smp_mb__after_atomic();
    if (cpdma_check_free_tx_desc(txch))
        netif_tx_wake_queue(txq);
}
```

Why?

If descriptor ring becomes full:

- Stop sending packets to this queue
- Prevent SKB flood
- Wait for TX interrupts to free descriptors
- Wake queue when there's space

This is standard Linux TX queue flow control.

cpdma_chan_submit() does the heavy lifting:

- Allocate a TX Buffer Descriptor (BD) from the channel's free list.
- Fill the BD with the physical address of skb->data, the length, and the destination EMAC port number (priv->emac_port).
- Set the OWN bit so the hardware knows it can fetch the descriptor.
- Advance the ring pointer and, if needed, "kick" the DMA engine (write to the channel's TX_PRI_REG).

❖ 9. Success path

```
return NETDEV_TX_OK;
```

Network stack knows transmission was accepted.

❖ 10. Failure path

If DMA submission failed:

fail:

```
ndev->stats.tx_dropped++;
netif_tx_stop_queue(txq);
...
if (cpdma_check_free_tx_desc(txch))
    netif_tx_wake_queue(txq);
```

```
return NETDEV_TX_BUSY;
```

- Drop stats increment
- Stop queue
- If space available → wake queue again
- Network stack retries the SKB later
- After a successful submit we double-check that at least one free descriptor remains. If not, we stop the kernel's TX queue to avoid flooding the driver with more skbs. The memory barrier guarantees that the stop is seen by other CPUs before we potentially re-enable the queue.

FINAL SUMMARY (Notes Version)

cpsw_ndo_start_xmit()

1. Extract driver private/context data
2. Ensure Ethernet min frame size (pad SKB)
3. Handle hardware TX timestamp request
4. Select correct TX queue (QOS / multiqueue support)
5. Choose CPDMA channel for that queue
6. Mark SKB for SW timestamp
7. Submit SKB to CPDMA (DMA descriptor)
8. If DMA ring is full → stop TX queue
9. Return success OR indicate busy

If you want next notes for:

- RX path (cpsw_rx_handler, cpsw_rx_poll)
- TX IRQ flow
- CPDMA descriptor format
- SKB life cycle in CPSW
- How ALE affects transmission

Tell me and I'll continue.

Summary

Thursday, December 11, 2025 4:30 PM

Below is a **clean, crisp, note-style theoretical summary** of what `cpsw_ndo_start_xmit()` does and **how** it works — no code, no complexity, just the logic.

SUMMARY — What `cpsw_ndo_start_xmit()` Does (Theoretical)

`cpsw_ndo_start_xmit()` is the **TX path** of the CPSW Ethernet driver.

It is called by the Linux networking stack **whenever a packet needs to be transmitted** out of the CPSW interface (eth0/eth1).

This function's job is:

1. Receive a packet (SKB) from the Linux stack

Linux passes a struct `sk_buff *skb` to the driver.

This skb holds:

- packet data
- metadata
- checksum info
- timestamp flags
- queue selection info

The driver must give this skb to the CPSW hardware for transmission.

2. Ensure the packet meets minimum Ethernet frame size

Ethernet requires a **minimum frame length** (usually 60 bytes + CRC).

If the packet is smaller:

- The driver pads it
- Ensures hardware will accept it

If padding fails → drop packet.

3. Handle hardware timestamp requests

If the packet asks for **hardware timestamping** (PTP/IEEE1588):

- Driver checks whether CPSW CPTS can timestamp it
- Marks the skb as “timestamp in progress”
- When hardware reports TX event, timestamp is attached

This is needed for precision time sync (industrial and automotive networks).

4. Select the TX queue

Linux supports **multi-queue TX** (multiple hardware TX channels).

`skb_get_queue_mapping(skb)` returns:

- “Which TX queue did the stack choose for this SKB?”

If the selected queue > number of CPSW hardware queues,

→ Driver remaps it safely.

Why?

Load distribution and QoS.

5. Find CPSW TX DMA channel

Using the queue index, the driver picks the **matching DMA channel**:

queue N → cpdma channel N

This channel will handle:

- DMA descriptor allocation
- Ownership transfer
- Hardware dispatch

6. Hand the SKB to CPDMA (submit descriptor)

Main operation:

Give packet buffer to CPSW DMA engine so it can transmit it over wire.

This creates a TX descriptor with:

- buffer address
- length
- port number
- flags

If descriptor submission fails:

- Stop TX queue
- Drop packet

7. Check if TX descriptors are exhausted

CPSW has a limited number of transmit descriptors.

If driver sees that **no TX descriptor is free**:

→ `netif_tx_stop_queue()`

Linux will stop giving more packets to this queue.

This prevents queue overflow and freezes.

When hardware finishes transmitting some packets:

→ Free descriptors

→ Driver wakes queue: `netif_tx_wake_queue()`

8. Return result to networking stack

- **NETDEV_TX_OK** → packet accepted
- **NETDEV_TX_BUSY** → hardware cannot accept packet now

This tells Linux whether it can continue sending or must wait.

Overall Flow (Very High Level)

Kernel wants to transmit



cpsw_ndo_start_xmit()



Verify packet & pad to min size



Check hardware timestamp requirements



Select TX queue and DMA TX channel



Submit packet to CPDMA for transmission



Check if TX descriptors exhausted



Possibly stop or wake TX queue



Return OK/BUSY status



Theoretical Description in One Paragraph

`cpsw_ndo_start_xmit()` is the CPSW driver's transmit entry point.

It takes packets from the Linux network stack, ensures they meet minimum Ethernet size, handles timestamping requests, selects the appropriate transmit queue, maps the packet into a hardware DMA descriptor, and submits it to CPSW CPDMA to be transmitted out of the physical port. If the hardware is out of descriptors, the driver temporarily stops the TX queue to avoid overflow, and restarts it when descriptors become available. The function then reports success or busy status back to the networking stack.

If you want next, I can give:

- **RX path summary (`cpsw_rx_handler`)**
- **CPDMA TX descriptor flow**
- **PTP timestamp flow**
- **Queue mapping and QoS explanation**

Just ask.

NDO_STOP

Thursday, December 11, 2025 4:42 PM

Below is a **clean, simple, notes-style explanation** of the part of `cpsw_ndo_stop()` that you posted.
No suggestions, no questions — only crisp theory.

Function: `cpsw_ndo_stop(struct net_device *ndev)`

Purpose:

This function is called when the network interface goes DOWN.

Examples when it is triggered:

- ifconfig eth0 down
- ip link set eth0 down
- System shutdown
- Driver unload
- Network namespace teardown

Its job is to **stop packet transmission, disable carrier, stop RX/TX engines, and clean up hardware state** to a safe condition.

Step-by-step theoretical explanation

1. Retrieve driver private structures

```
struct cpsw_priv *priv = netdev_priv(ndev);  
struct cpsw_common *cpsw = priv->cpsw;
```

- `cpsw_priv` = per-interface state (eth0 or eth1)
- `cpsw_common` = shared switch + DMA hardware block

The stop routine needs both because stopping the interface requires:

- disabling per-port MAC
- disabling shared DMA channels

2. Log shutdown event

```
cpsw_info(priv, ifdown, "shutting down cpsw device\n");
```

- Driver prints informational message
- Useful for debugging interface teardown

3. Handle multicast list cleanup

```
__hw_addr_ref_unsync_dev(&ndev->mc, ndev, cpsw_purge_all_mc);
```

Theory:

- Ethernet interfaces maintain a **multicast filter list** (e.g. for IPv6, mDNS).
- Hardware also stores multicast address entries (via ALE).

- When the device goes down, these must be removed.
- This call:
- Iterates all multicast addresses programmed in hardware
 - Removes them from CPSW ALE (Address Lookup Engine)
 - Ensures next time interface comes up, it starts clean

4. Stop transmission immediately

```
netif_tx_stop_all_queues(priv->ndev);
```

Theory:

- Linux can have multiple TX queues
- TX queues must be **stopped** so kernel stops sending SKBs to driver
- Prevents new packets being submitted to DMA

After this line:

- No new packets will enter ndo_start_xmit()
- Pending packets in hardware will eventually drain

5. Mark link as OFF

```
netif_carrier_off(priv->ndev);
```

Theory:

- Tells the networking stack: “link is down”
- Userspace tools (ip link, ifconfig) will show NO-CARRIER
- Upper layers (TCP, IPv6 autoconf, bridging) react accordingly

Even if the PHY is still sending a link-up signal,

the **net_device** is declared DOWN internally.

SUMMARY — (Theory only)

cpsw_ndo_stop() performs the required shutdown steps when an Ethernet interface is brought down:

1. **Announce shutdown** – logs the interface going down.
2. **Clean multicast configuration** – removes all MAC filters from hardware.
3. **Stop all transmit queues** – prevents the kernel from sending more packets.
4. **Turn off carrier** – tells networking stack the link is down.

Here is a **simple, clean, theory-only summary of what cpsw_ndo_stop() does and how it works**, written like **notes**, exactly as you requested.

cpsw_ndo_stop() — Theoretical Summary (What & How)

cpsw_ndo_stop() is called by the Linux networking stack when:

- The interface is brought down (ifconfig eth0 down, ip link set eth0 down)
- System shutdown/restart happens
- Driver is unloaded

This function **shuts down the CPSW networking hardware safely** so no more packets are received or transmitted.

★ Overall Purpose

The function:

5. **Stops traffic**
6. **Disables NAPI polling**
7. **Disables interrupts**
8. **Stops DMA engine**
9. **Stops ALE switch logic**
10. **Stops PHY slave ports**
11. **Releases hardware resources**
12. **Puts CPSW hardware block into low power**

★ Step-by-Step Theory

1. Print shutdown message

```
cpsw_info(priv, ifdown, "shutting down cpsw device\n");
```

Just logs “interface going down”.

2. Remove multicast filters

```
_hw_addr_ref_unsync_dev(&ndev->mc, ndev, cpsw_purge_all_mc);
```

Meaning:

- Remove all multicast addresses programmed into hardware.
- Ensures no stale multicast entries remain in switch/ALE.

3. Stop all TX queues

```
netif_tx_stop_all_queues(priv->ndev);
```

Meaning:

- Kernel will stop sending packets for transmission.
- Prevents tx queue overflow when shutting down.

4. Mark link as OFF

```
netif_carrier_off(priv->ndev);
```

Meaning:

- Tells networking core that the interface is not operational.
- Userspace tools will see carrier=false.

5. If this is the LAST active user (usage_count <= 1):

This part stops the **shared resources**, because CPSW can have:

- eth0

- eth1

Both may share the same hardware block, so resources shutdown only when both interfaces are down.

✓ Disable NAPI (no more polling)

```
napi_disable(&cpsw->napi_rx);  
napi_disable(&cpsw->napi_tx);
```

Stops RX/TX software polling.

✓ Unregister CPTS

```
cpts_unregister(cpsw->cpts);
```

Stops hardware timestamping engine.

✓ Disable interrupts

```
cpsw_intr_disable(cpsw);
```

Turns off RX/TX IRQ generation from hardware.

✓ Stop DMA controller

```
cpdma_ctlr_stop(cpsw->dma);
```

Stops packet DMA — absolutely important to prevent memory accesses after device shutdown.

✓ Stop ALE (Address Lookup Engine)

```
cpsw_ale_stop(cpsw->ale);
```

Disables CPSW's internal switch table and forwarding logic.

✓ Destroy XDP queues

```
cpsw_destroy_xdp_rxqs(cpsw);
```

Stops any XDP packet processing queues (if enabled).

6. Stop each slave port

```
for_each_slave(priv, cpsw_slave_stop, cpsw);
```

Meaning for every port (port0, port1):

- Stop PHY
- Disable GMAC port
- Stop MAC statistics
- Reset port-specific DMA settings

7. Re-split DMA resources if needed

```
if (cpsw_need_resplit(cpsw))  
    cpsw_split_res(cpsw);
```

Meaning:

- Adjust RX/TX channel assignment for consistency.
- Some SoCs need this due to hardware quirks.

8. Decrease hardware usage counter

```
cpsw->usage_count--;
```

Tracks how many netdev interfaces are active.

If 2 interfaces:

- eth0 down → usage_count = 1 → do not shut down DMA
- eth1 down → usage_count = 0 → full shutdown

9. Put CPSW hardware into runtime PM low-power state

```
pm_runtime_put_sync(cpsw->dev);
```

Meaning:

- Power down CPSW module (clocks off, power domain may turn off)
- Saves energy

★ Final Theory Summary (Short)

cpsw_ndo_stop():

- Stops TX queues
- Stops link
- Disables NAPI
- Disables interrupts
- Stops DMA
- Stops ALE switching
- Stops each CPSW port
- Cleans up XDP queues
- Updates usage counter
- Puts hardware into low power

It ensures **CPSW hardware is safely shut down**, with no DMA accessing memory and no interrupts firing.

★ If you want next notes:

I can continue with:

- cpsw_tx_interrupt()
- cpsw_rx_interrupt()
- cpsw_rx_poll() (NAPI explanation)
- cpsw_tx_poll()
- cpdma_chan_submit() flow
- cpsw_slave_open()
- cpsw_ale internals

Just tell me which one.

cpsw_rx_handler

Friday, December 12, 2025 12:35 PM

Important structs

1. struct page_pool — WHAT IT IS & WHY IT EXISTS

Purpose

A **page_pool** is a special memory allocator used by NIC drivers for RX. It gives *extremely fast recycling of pages* used for receiving packets.

Why needed?

Normal alloc_page() is *too slow* for 10G/40G NICs.

The page_pool:

- preallocates pages
- keeps them in a locked cache
- avoids page refcount overhead
- returns pages to the driver directly (zero-cost recycle)
- supports DMA mapping for NIC receive buffers

Where used?

- Many modern NIC drivers
- XDP RX path (critical)
- AF_XDP zero-copy mode

✓ Main Fields of struct page_pool (high-level)

```
struct page_pool {  
    struct pp_params params;      // pool configuration  
    struct ptr_ring ring;        // recycled pages  
    struct dma_map_ops *dma_ops;  // DMA mapping functions  
    struct device *dev;          // device doing DMA  
    u32 pages_state_hold_cnt;   // pages being held by consumers  
    bool depleted;              // pool temporarily empty  
    atomic_t pages_allocated;    // stats: how many pages currently
```

```
allocated
```

```
...
```

```
};
```

✓ Explanation of important fields

params

User-supplied settings such as:

- page size
- DMA direction (DMA_FROM_DEVICE)
- number of pages to keep
- destructor callback

Example NIC config:

```
params.pool_size = 1024;  
params.dma_dir  = DMA_FROM_DEVICE;  
params.flags    = PP_FLAG_DMA_MAP;
```

ring

A lockless pointer ring storing recycled pages.

NIC workflow:

1. NIC DMA writes packet into page
2. Driver builds skb or XDP buffer
3. Once user finishes, page goes back into ring
4. NIC reuses it for next RX

Zero allocation in steady state

pages_state_hold_cnt

Tracks pages temporarily held by:

- XDP programs
- skb fragments

- Userspace (AF_XDP)

Ensures pool does not free pages prematurely.

depleted

Flag that the pool ran out of pages → fallback to slow path.

pages_allocated

Number of pages currently allocated from system because ring was empty.

Used for debugging and monitoring.



3. struct xdp_buff — WHAT IT IS

struct xdp_buff is the core data structure for XDP (eXpress Data Path). It represents a packet *before SKB allocation*, stored directly in a page from page_pool.

This allows:

- zero-copy
- extremely fast packet inspection
- running XDP programs
- dropping/redirectiong packets without touching SKB

✓ Full structure (simplified)

```
struct xdp_buff {
    void *data;      // start of packet data
    void *data_end; // end of packet data
    void *data_meta; // metadata space (optional)
```

```
struct xdp_rxq_info *rxq; // which RX queue this packet came from  
void *data_hard_start; // start of actual buffer (including  
headroom)  
};
```

✓ Explanation of each field

data

Pointer to the first byte of packet payload.

Equivalent to:

skb->data

data_end

Pointer to end of packet.

XDP program must always obey:

if (data + sizeof(header) > data_end) → reject packet

This avoids buffer overflows.

data_meta

Used by XDP programs for storing:

- metadata
- timestamps
- RSS hashes
- custom fields

data_hard_start

Points to beginning of buffer allocated by NIC via page_pool.

Layout:

[reserved headroom | XDP frame | tailroom]

↑ data

↑ data_end

NIC drivers place XDP headroom (usually 256 bytes) here.

rxq

Points to struct xdp_rxq_info.

Contains:

- RX queue index
- Device reference
- NAPI context
- Memory type (page_pool / skb / frag)

Allows XDP program to perform:

- redirect
- transmit
- map packet to correct queue

Below is a **clean, deep, kernel-internals-level explanation of struct page** exactly the way the Linux kernel uses it for:

- memory management
- networking (SKB fragments, XDP, page pools)
- DMA and NIC drivers
- high-performance RX/TX buffer handling

This is **NOT a shallow explanation** — this is the level driver writers must know.

★ What is struct page in Linux?

struct page is **the fundamental data structure representing a physical memory page** inside the Linux kernel.

Whenever the kernel allocates RAM, DMA buffers, SKB fragments, XDP

frame memory, page-pool memory, or filesystem pages — internally it is always dealing with a struct page.

A page is typically **4096 bytes (4 KB)** on most systems, but may be 64KB or others depending on architecture.

★ Why drivers **MUST** understand struct page?

NIC drivers use pages for:

- RX buffers
- XDP zero-copy
- Recycling memory through page_pool
- Passing fragments to SKBs
- DMA mapping

When your NIC receives a packet into memory → it is very likely going into memory represented by a struct page.

★ THE STRUCTURE ITSELF

Defined in:

include/linux/mm_types.h

The real struct is huge and full of unions, but conceptually it looks like:

```
struct page {  
    unsigned long flags;  
    atomic_t _refcount;  
    struct address_space *mapping;  
    pgoff_t index;  
  
    union {  
        struct {  
            unsigned long private;  
            struct list_head lru;  
        }  
    }  
};
```

```

};

struct {
    void *freelist;
    union {
        unsigned long counters;
        struct refcount_struct _mapcount;
    };
};

struct { ... slab fields ... };
struct { ... page-pool / network fields ... };
};

void *virtual; // only for HIGHMEM pages
};

```

The real struct has many more fields.

Let's explain the important ones **from a network-driver perspective**.

★ ➤ 1. flags

Bitmask describing properties of the page:

Examples:

Flag	Meaning
PG_locked	page is locked (no eviction)
PG_slab	page belongs to SLAB allocator
PG_reserved	do not free
PG_dma	suitable for DMA
PG_referenced	recently accessed

NIC drivers rarely look at flags directly but **memory allocators and DMA-mapping code depend on these flags**.

★ ➤ 2. refcount

Reference count → number of users of this page.

This is critical.

Example network usage:

- RX packet arrives into page
- Page is attached to XDP frame or SKB
- refcount++
- When SKB/XDP is freed → refcount--
- If refcount == 0 → page can be reused or freed

If you get refcount wrong → **memory corruption**, double-free, page leak.

This is why page_pool uses special APIs to manage refcount safely.

★ ➤ 3. mapping and index

These are used for filesystem and page cache.

Networking drivers do **not** use these.

★ ➤ 4. The huge union inside

This is the most complex part.

Same physical struct page is reused for many subsystems:

- SLAB allocator
- Buddy allocator
- Page cache
- Page pool (networking)
- Anonymous memory
- Compound pages (huge pages)

Networking particularly cares when the page is used for:

- page allocator (buddy),
- page_pool (packet RX),
- SKB shared fragments,

→ XDP memory.

Inside the union you'll find:

4A. Iru

Used when the page is on an LRU list (page cache, reclaim).

Page-pool often removes pages from LRU to avoid reclaiming RX buffers.

4B. private

Used by many subsystems to store custom metadata.

NIC drivers + page_pool store metadata here such as:

- DMA address
- which pool it belongs to
- “page_pp_magic” for debugging

4C. freelist

Used by SLUB for slab pages; rarely used in networking.

4D. _mapcount

How many PTEs map this page.

Not used directly in NIC drivers.

★ ➤ 5. virtual pointer

Only present on HIGHMEM systems.

Used to get kernel-virtual address of the page.

On modern 64-bit systems most memory is directly mapped, so page_address(page) simply returns a valid pointer.

Used by NIC drivers to pass the **CPU-accessible pointer** to DMA buffers.

★ How networking uses struct page

1. RX Buffers

NIC DMA engine writes packet data directly into memory areas backed by struct page.

Flow:

NIC DMA → physical_addr → struct page → kernel virtual addr → XDP/
SKB

NIC drivers often allocate pages via:

```
page = alloc_pages_node();
```

Then:

```
dma_addr = dma_map_page(dev, page, offset, size,  
DMA_FROM_DEVICE);
```

2. SKB Fragments (frags[])

SKB stores packet data in:

- Linear data (skb->data)
- Fragment list (skb_shinfo(skb)->frags[])

Each frag points to a struct page.

Example:

```
skb_frag_t *frag = &skb_shinfo(skb)->frags[0];  
struct page *page = skb_frag_page(frag);
```

The NIC driver fills these frags with pages from page_pool or
alloc_pages().

3. XDP Data

xdp_buff stores packet metadata for XDP. It points to memory inside a
struct page.

When the RX driver receives a packet, it fills:

xdp->data ← start of packet

xdp->data_end ← end of packet

xdp->data_hard_start ← page buffer start

And XDP returns one of:

- XDP_PASS → convert to SKB
- XDP_DROP → page is recycled
- XDP_TX → page is used for transmission
- XDP_REDIRECT → page sent to another interface or CPU

All uses go through struct page.

4. Page Pool

Page pool is an optimized allocator for NIC RX buffers.

struct page contains small metadata fields modified by page_pool.

page_pool uses:

- refcount
- private
- DMA address stored inside the page
- page recycling path

This gives zero-copy recycling without going through buddy allocator.

★ Why struct page is complex?

Because it must represent all types of memory:

- File-backed pages
- Anonymous memory pages
- Slab pages
- Huge pages
- Device DMA buffers
- Networking XDP pages
- Kernel internal memory

So it uses **many unions** so that different subsystems reuse the same memory area for their own metadata.

★ Visual Summary Diagram

struct page

```
└─ flags      → Page properties
└─ _refcount   → Reference count
└─ mapping, index → File / page cache binding
└─ union {
    └─ private   → NIC driver / page_pool metadata
    └─ lru       → linked list for cache
    └─ freelist   → slab allocator
    └─ counters   → mapcount or slab metadata
    └─ ...
}
└─ virtual     → kernel VA for HIGHMEM
```

★ Simplified Summary (Notebook Style)

struct page represents ONE physical page of memory.

Contains:

- **Status flags**
- **Reference counter**
- **DMA and NIC metadata**
- **Page cache linkage**
- **SLAB allocator metadata**
- **Page pool metadata**
- **SKB/XDP buffer memory mapping**

Used everywhere:

- RX/TX packet buffers
- Page pool
- SKB fragments
- XDP buffers
- DMA memory
- General memory allocation

If you want, I can give similar deep explanations for:

- ✓ struct page_pool
- ✓ struct page_frag_cache
- ✓ struct page_alloc
- ✓ struct xdp_buff
- ✓ struct skb_shared_info

Just tell me which one.

Code explain

Friday, December 12, 2025 3:04 PM

366-399

CPSW RX Handler: Port selection + Safe page reuse

1. Identify the RX port

- CPDMA embeds port number in RX descriptor.
- Driver uses CPDMA_RX_SOURCE_PORT(status) to extract.
- In dual-emac mode, this selects correct net_device (eth0 or eth1).

2. Validate RX status

- If status < 0 → descriptor indicates error → drop packet.
- If interface is down → do not pass packet to stack.

3. Dual EMAC special rule

If a packet arrives for a port that is DOWN **but** CPSW is still active through the other port:

- We **must not free** the page.
- Instead we **requeue** it to the RX DMA so descriptor count stays correct.

4. If safe to drop

If requeue conditions are NOT met:

- Recycle the page back to the page pool using
page_pool_recycle_direct()

Why?

- Prevent RX ring depletion
- Maintain DMA descriptor count
- Ensure high-performance zero-copy RX
- Support dual independent Ethernet interfaces using shared DMA

399

```
new_page = page_pool_dev_alloc_pages(pool);
if (unlikely(!new_page)) {
    new_page = page;
```

```
    ndev->stats.rx_dropped++;
    goto requeue;
}
```

Goal

For every RX packet received:

- The *old page* (with packet data) is handed to upper layers (or XDP).
- A *fresh page* must be supplied to CPDMA to refill the RX descriptor.
[Why allocate a fresh page?](#)
- RX is zero-copy.
- DMA buffers must always be available.
- Page_pool supports fast recycling.

Failure Cases

If allocation fails:

- System is under memory pressure.
- Page_pool depleted temporarily.

In that case:

- Reuse the same page (new_page = page).
- Count the drop (rx_dropped++).
- Requeue back to CPDMA.

Driver tries to avoid starving RX descriptors at all cost.

400-426

✿ What is happening here? (Expanded Notes)

1. A new packet arrives into a DMA memory page.

CPDMA has already copied raw Ethernet bytes inside a page.

2. Driver sets up an XDP buffer describing only the packet, not the whole page.

It tells XDP:

- where the packet starts
- where it ends
- what the original memory area is

It also hides any CPSW VLAN encapsulation.

3. The XDP buffer is fully prepared:

- `xdp.data` = packet start
- `xdp.data_end` = packet end
- `data_hard_start` = page start
- `rxq` = RX queue info
- `frame_sz` = max safe page size

4. The XDP program runs.

The XDP program can:

- drop
- redirect
- transmit
- abort
- or pass the packet to the kernel

5. If XDP drops/redirects/transmits → Driver stops here and requeues the buffer.

The packet **never reaches the kernel networking stack.**

6. If XDP passes the packet → Driver continues with normal skb creation.

XDP might have:

- changed the packet start
- removed headers
- added metadata

So driver recomputes headroom and proceeds.

Below are **clean, structured NOTES** explaining this **exact XDP handling block** in the CPSW RX path.

No questions, no suggestions — just **what it does, why it exists, and how it works**, step by step.

CONTEXT (VERY IMPORTANT)

This code runs in the **RX handler**, after:

- CPDMA has DMA'd a received Ethernet frame into a **page**
- status, len, page, and pa (page address) are already known
- The driver is deciding whether to:
- run **XDP**, or
- skip XDP and go directly to **skb creation**

This block executes **only if an XDP program is attached**.

```
if (priv->xdp_prog) {
```

```
    ...
```

```
}
```

So everything below is about **preparing and running XDP on a CPSW RX packet**.

1. WHY THIS BLOCK EXISTS

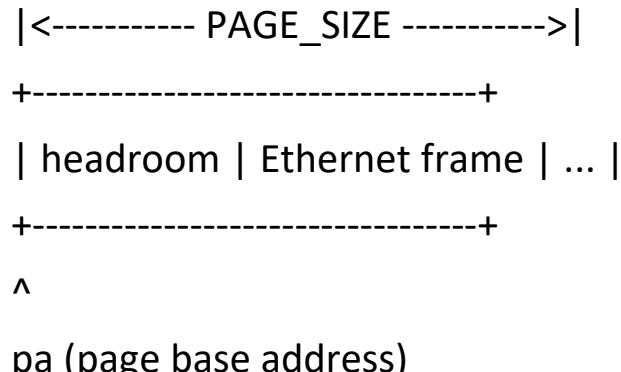
CPSW supports **XDP (eXpress Data Path)**, which means:

- Packets can be processed **before skb allocation**
- Packet handling happens **in page memory**, not skb memory
- Very fast (used for DDoS filtering, forwarding, drop, redirect)

To do this, the driver must convert a **DMA page** into an **xdp_buff**.

2. DMA MEMORY LAYOUT (CRITICAL TO UNDERSTAND)

When CPSW receives a packet, memory looks like this:



- pa = base address of the DMA page
- CPSW_HEADROOM = space reserved before packet
- Packet data starts **after headroom**
- VLAN encapsulation may add extra bytes

3. HANDLING VLAN ENCAPSULATION

```
if (status & CPDMA_RX_VLAN_ENCAP) {
```

What this means

- CPSW hardware may **strip VLAN tags** and place them in a **separate header**
- This is called **VLAN encapsulation**
- XDP must see the **real L2/L3 packet**, not the CPSW-specific header

Case 1: VLAN encapsulation present

```
xdp.data = pa + CPSW_HEADROOM +
           CPSW_RX_VLAN_ENCAP_HDR_SIZE;
xdp.data_end = xdp.data + len -
               CPSW_RX_VLAN_ENCAP_HDR_SIZE;
```

What is happening

- Skip:
 - CPSW headroom
 - CPSW VLAN encapsulation header
 - Point xdp.data to **actual Ethernet frame**
 - Adjust data_end so XDP sees correct packet length
- Why this is required*

XDP programs expect:

xdp.data → Ethernet header

They must NOT see CPSW-specific metadata.

Case 2: No VLAN encapsulation

```
xdp.data = pa + CPSW_HEADROOM;
```

```
xdp.data_end = xdp.data + len;
```

Meaning

- Packet starts immediately after headroom
- Normal Ethernet frame
- No adjustment needed

4. METADATA HANDLING

```
xdp_set_data_meta_invalid(&xdp);
```

What this does

- Marks **no XDP metadata** is present
- Prevents accidental access to invalid memory
- Required for correctness and verifier safety

5. HARD START ADDRESS

```
xdp.data_hard_start = pa;
```

Meaning

- This is the **earliest valid memory address**
- XDP helpers (like bpf_xdp_adjust_head) must not move data before this
- Protects headroom and DMA safety

6. RX QUEUE ASSOCIATION

```
xdp.rxq = &priv->xdp_rxq[ch];
```

Why this exists

- Each RX channel has its own **XDP RX queue**
- Required for:
- XDP redirect
- AF_XDP sockets
- Multi-queue awareness

This links the packet to **which RX channel** it came from.

7. FRAME SIZE DECLARATION

```
xdp.frame_sz = PAGE_SIZE;
```

Meaning

- Entire DMA page is available to XDP
- Needed for:
- bounds checking
- head adjustments
- verifier correctness

8. PORT CALCULATION (DUAL EMAC SUPPORT)

```
port = priv->emac_port + csw->data.dual_emac;
```

Why this matters

- CPSW can run in **dual EMAC mode**
- Each packet must be associated with a **hardware port**
- XDP programs may use port info (e.g., forwarding decisions)

9. RUNNING THE XDP PROGRAM

```
ret = csw_run_xdp(priv, ch, &xdp, page, port, &len);
```

What this function does internally

- Calls `bpf_prog_run_xdp()`
- Executes attached BPF program
- XDP program can:
- inspect headers
- modify packet
- change packet length
- decide packet fate

10. HANDLING XDP RETURN VALUE

```
if (ret != CPSW_XDP_PASS)  
    goto requeue;
```

Meaning of return values

XDP Result	Action
XDP_DROP	Packet dropped

XDP_ABORTED	Drop + error
XDP_TX	Transmit back out
XDP_REDIRECT	Redirect to another interface / AF_XDP
XDP_PASS	Continue normal RX (skb creation)

Only **XDP_PASS** allows the packet to continue.

All others **skip skb creation** and requeue the page.

11. RECALCULATING HEADROOM

```
headroom = xdp.data - xdp.data_hard_start;
```

Why this is needed

- XDP program may have:
- pulled headers
- pushed headers
- modified packet start
- Driver must compute **new headroom**
- This is later used during skb construction

12. CLEAR VLAN ENCAP FLAG

```
status &= ~CPDMA_RX_VLAN_ENCAP;
```

Why this is critical

- XDP program may:
- modify VLAN tags
- insert/remove VLAN headers
- CPSW VLAN encapsulation info is now **invalid**
- Driver must not rely on hardware VLAN metadata anymore

FINAL FLOW SUMMARY (RX + XDP)

DMA receives packet into page



Driver prepares xdp_buff



Fix VLAN abstraction

```
↓  
Set xdp buffer pointers  
↓  
Run XDP program  
↓  
If XDP_DROP / TX / REDIRECT → stop  
↓  
If XDP_PASS → build skb and continue normal RX
```

KEY TAKEAWAYS (IMPORTANT)

- **XDP runs before skb exists**
- **DMA page is reused**
- **No memory copy**
- **VLAN handling must be fixed before XDP**
- **Only XDP_PASS reaches Linux networking stack**
- **All others bypass skb completely**

This block is the **bridge between CPDMA RX and XDP**, and it is one of the most performance-critical paths in the CPSW driver.

- Below are **clean, structured NOTES** explaining **exactly what this RX → SKB creation block does, why each step exists, and how it fits into CPSW RX flow.**
- No questions, no suggestions — only explanation.
-
-
-  **CONTEXT (where this code sits)**
- CPSW RX DMA has delivered a packet into a **DMA-backed page**
- XDP processing has either:
- **not been enabled**, or
- **returned XDP_PASS**
- Now the driver must **convert raw DMA memory → Linux sk_buff**
- This is the **handoff point from driver to Linux network stack**
-
-

- ◇ CODE BLOCK (RX → SKB path)
- /* pass skb to netstack if no XDP prog or returned XDP_PASS */
- skb = build_skb(pa, cpsw_rxbuf_total_len(pkt_size));
- if (!skb) {
 - ndev->stats.rx_dropped++;
 - page_pool_recycle_direct(pool, page);
 - goto requeue;
- }
-
- skb_reserve(skb, headroom);
- skb_put(skb, len);
- skb->dev = ndev;
-
- if (status & CPDMA_RX_VLAN_ENCAP)
 - cpsw_rx_vlan_encap(skb);
-
- if (priv->rx_ts_enabled)
 - cpts_rx_timestamp(cpsw->cpts, skb);
-
-
- skb->protocol = eth_type_trans(skb, ndev);
-
-
- ① build_skb() — Create SKB from DMA memory
- skb = build_skb(pa, cpsw_rxbuf_total_len(pkt_size));
- What this does
- Creates an sk_buff **without copying packet data**
- Uses the **existing DMA page memory**
- pa = pointer to DMA buffer (page-backed)
- Zero-copy RX path
- Why this matters
- Avoids memcpy() → **high performance**

- DMA page becomes SKB data buffer
- Page lifetime now managed by SKB
- If it fails
- `ndev->stats.rx_dropped++;`
- `page_pool_recycle_direct(pool, page);`
- goto requeue;
-
- Packet dropped
- DMA page returned to page-pool
- RX descriptor reused
-
-
- **2** `skb_reserve()` — Skip driver/XDP headroom
- `skb_reserve(skb, headroom);`
- What this does
- Moves `skb->data` forward by headroom bytes.
- Why headroom exists
- DMA buffer layout:
- | hard_start | headroom | Ethernet frame | tailroom |
- Headroom is reserved for:
- CPSW internal alignment
- XDP modifications
- VLAN adjustments
- Potential encapsulation headers
- Result
- `skb->data` now points to **Ethernet header**
- Stack sees a clean L2 frame
-
-
- **3** `skb_put()` — Set packet length
- `skb_put(skb, len);`
- What this does
- Advances `skb->tail`

- Sets valid packet size
- After this:
- `skb->data` → Ethernet header
- `skb->tail` → end of frame
- `skb->len` = len
-
- This is the **actual RX packet length**.
-
-
- **4 Assign network device**
- `skb->dev = ndev;`
- Why needed
- Tells kernel **which interface received this packet**
- Used for:
- RX statistics
- Routing decisions
- Netfilter hooks
- BPF programs
-
-
- **5 VLAN encapsulation handling (CPSW specific)**
- `if (status & CPDMA_RX_VLAN_ENCAP)`
- `cpsw_rx_vlan_encap(skb);`
-
- What this means
- CPSW hardware can:
- Strip VLAN tag from frame
- Deliver VLAN info separately
- Set CPDMA_RX_VLAN_ENCAP flag
- This function:
- Rebuilds standard VLAN header
- Adjusts SKB so Linux sees a **normal 802.1Q frame**
- Result:
- Ethernet → VLAN → EtherType → Payload

-
- **6 RX hardware timestamping (CPTS)**
- if (priv->rx_ts_enabled)
- cpts_rx_timestamp(cpsw->cpts, skb);
-
- What this does
- Reads RX timestamp from CPSW CPTS hardware
- Attaches it to SKB metadata
- Used by:
- PTP / IEEE 1588
- Time-sensitive networking (TSN)
- Timestamp stored in:
- skb_shared_hwtstamps
-
- **7 eth_type_trans() — Hand packet to Linux stack**
- skb->protocol = eth_type_trans(skb, ndev);
- What this function does
- Parses Ethernet header
- Determines protocol:
- IPv4
- IPv6
- ARP
- VLAN
- Sets:
- skb->protocol
- Adjusts skb->data to point to **L3 header**
- Example
- Before:
- skb->data → Ethernet header
- After:
- skb->data → IP header

- skb->protocol = ETH_P_IP
-
-
-  AFTER THIS POINT
- The driver will call:
- napi_gro_receive(skb) or netif_receive_skb(skb)
- And the packet enters:
- L2 → L3 → L4 → Socket
-
-  COMPLETE FLOW SUMMARY
- RX path (no XDP or XDP_PASS)
- DMA page
- ↓
- build_skb() (zero-copy)
- ↓
- skb_reserve() (skip headroom)
- ↓
- skb_put() (set length)
- ↓
- VLAN fixup
- ↓
- RX timestamp
- ↓
- eth_type_trans()
- ↓
- Linux networking stack
-
-
-  KEY TAKEAWAYS
- **No data copy** → DMA page reused

- SKB built directly on DMA memory
- VLAN + timestamp handled **before** stack
- Clean separation:
- **Driver work ends here**
- **Stack work begins here**
-
-
- This block is the **exact boundary between CPSW hardware RX and Linux networking stack.**
-

summary

Monday, December 15, 2025 10:55 AM

- Below are **clean NOTES**, exactly in the style you asked for.
No questions, no suggestions — only **what happens, which function, and what it does.**

CPSW RX PATH (WITH XDP) — STEP-BY-STEP NOTES

This is the **exact logical flow** of RX handling in the CPSW driver, with **required function calls** and **what each one does.**

1 RX DMA WRITES PACKET INTO MEMORY PAGE

Hardware action

- **CPDMA RX engine** receives an Ethernet frame from PHY
- DMA writes packet data into a **pre-mapped page** provided earlier by driver
- RX descriptor status is updated

Driver entry

RX interrupt / NAPI poll calls the **RX handler**:

`cpsw_rx_handler()`

2 DRIVER READS CPDMA STATUS

Status contains:

- packet length
- source port (dual-emac)
- VLAN encapsulation bit
- error flags

Macro used

`CPDMA_RX_SOURCE_PORT(status)`

Purpose

- Identify **which CPSW slave port** the packet arrived on

3 SELECT CORRECT net_device (DUAL-EMAC MODE)

Code logic

```
if (cpsw->data.dual_emac && status >= 0) {
    port = CPDMA_RX_SOURCE_PORT(status);
    ndev = cpsw->slaves[port - 1].ndev;
}
```

What this does

- Maps RX packet to correct interface (eth0 or eth1)
- Required because **one RX DMA channel serves both ports**

4 GET PAGE POOL FOR RX CHANNEL

Function / data

```
pool = cpsw->page_pool[ch];
```

What this does

- Gets the **page_pool** managing RX pages
- Ensures:
- fast allocation
- DMA-safe memory
- zero-copy reuse

5 HANDLE DOWN INTERFACE / ERROR CASES

If interface is DOWN or RX error

Functions used:

```
page_pool_recycle_direct(pool, page);
```

What this does

- Returns page to pool
- Prevents RX descriptor starvation
- Keeps DMA running

6 ALLOCATE A NEW RX PAGE

Function

```
new_page = page_pool_dev_alloc_pages(pool);
```

What this does

- Allocates a **replacement page**
- Needed because current page may go to stack or XDP Failure case

If allocation fails:

```
ndev->stats.rx_dropped++;
```

```
new_page = page;
```

7 PREPARE XDP BUFFER (IF XDP ENABLED)

Condition

```
if (priv->xdp_prog)
```

XDP buffer fields set:

a) Data pointers

```
xdp.data    = packet start
```

```
xdp.data_end = packet end
```

Handled differently if VLAN encapsulation exists.

b) Metadata

```
xdp_set_data_meta_invalid(&xdp);
```

c) Hard start

```
xdp.data_hard_start = pa;
```

d) RX queue

```
xdp.rq = &priv->xdp_rxq[ch];
```

e) Frame size

```
xdp.frame_sz = PAGE_SIZE;
```

What this does

- Constructs an **XDP-visible packet**
- No skb yet
- Raw packet memory exposed to eBPF

8 RUN XDP PROGRAM

Function

```
ret = cpsw_run_xdp(priv, ch, &xdp, page, port, &len);
```

What this does

- Executes attached **XDP eBPF program**
- Program may:
 - modify packet
 - drop
 - redirect
 - transmit
 - pass to stack

9 HANDLE XDP RETURN VALUE

Possible return values

Return	Meaning	Action
--------	---------	--------

XDP_DROP	Drop packet	recycle page
XDP_TX	Transmit back	driver handles
XDP_REDIRECT	Redirect to other iface/CPU	done
XDP_ABORTED	Error	recycle
XDP_PASS	Go to normal stack	continue

Code path

```
if (ret != CPSW_XDP_PASS)
    goto requeue;
```

10 CALCULATE HEADROOM AFTER XDP

Code

```
headroom = xdp.data - xdp.data_hard_start;
```

Why needed

- XDP may adjust packet start
- skb must match new layout

1 1 BUILD skb (NORMAL NETWORK STACK)

Function

```
skb = build_skb(pa, cpsw_rxbuf_total_len(pkt_size));
```

What this does

- Wraps existing DMA page into an skb
- **Zero copy**
- No data copy involved

1 2 ADJUST skb CONTENTS

Reserve headroom

```
skb_reserve(skb, headroom);
```

Add payload length

```
skb_put(skb, len);
```

Assign device

```
skb->dev = ndev;
```

1 3 HANDLE VLAN & TIMESTAMP

VLAN

```
cpsw_rx_vlan_encap(skb);
```

Timestamp (PTP)

```
cpts_rx_timestamp(cpsw->cpts, skb);
```

1 4 SET PROTOCOL & HAND OVER TO STACK

Protocol detection

```
skb->protocol = eth_type_trans(skb, ndev);
```

Deliver packet

```
netif_receive_skb(skb);
```

What this does

- Packet enters Linux network stack
- IP / TCP / UDP processing begins

1 5 UPDATE RX STATISTICS

Counters

```
ndev->stats.rx_packets++;
```

```
ndev->stats.rx_bytes += len;
```

1 6 RELEASE OLD PAGE

Function

```
page_pool_release_page(pool, page);
```

What this does

- Marks page as consumed
- Prevents double reuse

1 7 PREPARE METADATA FOR NEW RX PAGE

Metadata stored

```
xmeta->ndev = ndev;
```

```
xmeta->ch = ch;
```

Why

- Needed when RX page returns from DMA
- Identifies RX context

1 8 REQUEUE RX DESCRIPTOR TO CPDMA

DMA address

```
dma = page_pool_get_dma_addr(new_page) + CPSW_HEADROOM;
```

Submit descriptor

```
cpdma_chan_submit_mapped(rx_ch, new_page, dma, pkt_size, 0);
```

What this does

- Gives CPDMA a **fresh RX buffer**
- Keeps RX engine running continuously

FINAL RX FLOW SUMMARY (FUNCTION-ORIENTED)

RX DMA → page

↓

cpsw_rx_handler()

↓

CPDMA_RX_SOURCE_PORT()

↓

page_pool_dev_alloc_pages()

↓

Prepare xdp_buff

↓

cpsw_run_xdp()

↓

[DROP / TX / REDIRECT] → recycle

↓

build_skb()

↓

eth_type_trans()

↓

netif_receive_skb()

↓

page_pool_release_page()

↓

cpdma_chan_submit_mapped()

This is the **complete RX logic** of CPSW with **XDP + page_pool + CPDMA**, explained as **driver notes**.

If you want, next we can convert this into:

- **flowchart**
- **one-page exam notes**
- **comparison with non-XDP RX**
- **minimal RX driver skeleton**

Just say the word.

Got it — I will keep the explanation **medium level**, clear, simple, and not overloaded.

Let's retry: **What is MDIO / MII / MDIO bus** at a level that a new person can understand and still relate to your driver code.

Medium-Level Explanation (Not too deep, not too shallow)

1. What is a PHY?

- The PHY is the chip that handles **actual Ethernet signals** on the cable.
- It decides link speed (10/100/1000), duplex, auto-negotiation.
- It sits outside the CPU/SoC or sometimes inside.

The driver must **talk to the PHY** to:

- Read link status
- Configure speed/duplex
- Reset it
- Start auto-negotiation

2. What is the MAC?

- MAC is inside the SoC (your CPMAC).
- It deals with:
- Sending / receiving Ethernet frames
- DMA
- Packet buffers

MAC \leftrightarrow (wire interface) \leftrightarrow PHY \leftrightarrow Cable

3. What is MII?

- MII = a simple interface connecting **MAC to PHY**.
- Only sends/receives **digital data** (not analog signals).

You don't program MII directly.

The hardware block just uses it.

★ 4. What is MDIO?

- MDIO = *Management Data Input/Output*.
- A tiny **two-wire bus** (MDC clock + MDIO data).
- Used **ONLY to read/write registers inside PHY**.

MDIO lets the MAC ask:

- “Is link up?”
- “What speed?”
- “What is PHY ID?”
- “Start or stop auto-negotiation”

So MDIO is like a **control channel** to the PHY.

★ 5. What is MDIO Bus (`mii_bus`) in Linux?

Linux needs a way to manage all PHYs.

So it has a structure called **struct mii_bus** which represents the MDIO controller.

This structure stores:

- `read()` → how to read a PHY register
- `write()` → how to write
- `reset()` → how to reset MDIO
- `phy_mask` → which PHY addresses to ignore
- `priv` → pointer to MDIO hardware registers

When you call:

```
mdiobus_register(cpmac_mii);
```

Linux will:

1. Scan PHY addresses (0–31)
2. Detect which PHY is present
3. Create `phy_device` objects

Without MDIO bus → **driver cannot detect PHY → Ethernet won't work.**

★ 6. What is happening in your cpmac_init()? (Very simple view)

Step-by-step simplified:

(1) Allocate MDIO bus

```
cpmac_mii = mdiobus_alloc();
```

Creates a Linux object for MDIO communication.

(2) Fill callbacks

```
cpmac_mii->read = cpmac_mdio_read;  
cpmac_mii->write = cpmac_mdio_write;
```

Tell Linux **how** to read/write PHY registers.

(3) Map hardware MDIO registers

```
cpmac_mii->priv = ioremap(...);
```

So driver can control MDIO hardware.

(4) Reset hardware blocks

```
ar7_device_reset(...);
```

Bring CPMAC + PHY out of reset so MDIO works.

(5) Detect PHY present

```
mask = cpmac_read(... ALIVE register ...);
```

This tells which PHY addresses responded.

(6) Set PHY mask

```
cpmac_mii->phy_mask = ~(mask | 0x80000000);
```

Tell Linux which PHYs to ignore.

(7) Register MDIO bus

```
mdiobus_register(cpmac_mii);
```

This triggers PHY detection.

(8) Register CPMAC platform driver

```
platform_driver_register(&cpmac_driver);
```

Now the MAC driver can probe.

★ 7. Summary in one sentence

MDIO/MDIO bus = the way MAC talks to PHY.

MII = the MAC-PHY data interface.

mii_bus is the Linux representation of MDIO.

If this level is good, paste the **next function**, and I will keep the explanation at the same medium level.

Perfect.

Now I will explain these **CPMAC registers in a medium, simple, beginner-friendly way**, not overloaded, not too high-level.

✓ VERY IMPORTANT: What these registers belong to

These registers are **NOT PHY registers**.

They are **MAC registers** (internal Ethernet controller inside the SoC).

- **PHY registers** → accessed via **MDIO**
- **MAC registers** → accessed via **memory-mapped I/O** (ioremap + readl writel)

Your list is **MAC (CPMAC) hardware registers** used by the Linux driver.

✓ I will explain the registers in groups to make it simple

◇ 1. TX / RX CONTROL

These control the **transmit and receive engines** of the MAC.

CPMAC_TX_CONTROL (0x0004)

- Turn TX (transmit) ON or OFF
- If 1 → MAC can start sending packets

- If 0 → TX is disabled
`CPMAC_TX_TEARDOWN (0x0008)`
- Used to **stop TX DMA channel safely**
- Clears descriptors
- Used when interface goes DOWN

`CPMAC_RX_CONTROL (0x0014)`

- Turn RX (receive) ON or OFF
- Enable MAC to accept packets
`CPMAC_RX_TEARDOWN (0x0018)`
- Safely shut down RX DMA
- Stops DMA ring processing

◇ 2. MBP (Multicast/Broadcast/Promiscuous) CONTROL

`CPMAC_MBP (0x0100)` controls **packet filtering**:

This decides **which packets the MAC should accept**.

Bit explanation:

- MBP_RXPASSCRC – allow frames with CRC included
- MBP_RXQOS – enable QoS
- MBP_RXNOCHAIN – do not chain buffers
- MBP_RXCMF – copy MAC control frames
- MBP_RXSHORT – accept short frames
- MBP_RXCEF – accept error frames
- MBP_RXPROMISC – **promiscuous mode**, accept EVERYTHING
- MBP_RXBCAST – accept broadcast packets
- MBP_RXMCAST – accept multicast packets

Linux driver sets these depending on:

- interface mode
- user commands (promiscuous mode, all-multicast, etc.)

◇ 3. MAC UNICAST / MULTICAST ADDR FILTERS

These registers configure **which MAC addresses** the controller should listen to.

`CPMAC_UNICAST_ENABLE`

Enable unicast channels

`CPMAC_UNICAST_CLEAR`

Clear unicast filters

`CPMAC_MAC_ADDR_LO(channel)`

The actual MAC address for channel N (bottom 32 bits)

`CPMAC_MAC_ADDR_MID / HI`

Upper bits of MAC address

Hash tables:

- `CPMAC_MAC_HASH_LO`
- `CPMAC_MAC_HASH_HI`

Used for multicast filtering.

◇ 4. MAC CONTROL REGISTER

`CPMAC_MAC_CONTROL` (0x0160) controls **MAC operation mode**.

Bits:

- `MAC_TXPTYPE` – packet type
- `MAC_TXPACE` – pacing
- `MAC_MII` – using MII mode
- `MAC_TXFLOW` – enable TX flow control
- `MAC_RXFLOW` – enable RX flow control
- `MAC_MTEST` – test mode
- `MAC_LOOPBACK` – internal MAC loopback
- `MAC_FDX` – **full duplex**

Driver updates this depending on PHY link settings:

- full/half duplex
- speed
- flow control

◇ 5. MAC STATUS REGISTER

`CPMAC_MAC_STATUS` (0x0164)

Shows:

- Pause frames
- Flow control
- QoS enable

Not commonly used in simple drivers.

◇ 6. INTERRUPT REGISTERS

The MAC has **TX / RX interrupts**, and general MAC interrupt.

TX interrupt registers:

- CPMAC_TX_INT_ENABLE
- CPMAC_TX_INT_CLEAR

RX interrupt registers:

- CPMAC_RX_INT_ENABLE
- CPMAC_RX_INT_CLEAR

Global interrupt:

- CPMAC_MAC_INT_VECTOR
- CPMAC_MAC_INT_ENABLE
- CPMAC_MAC_INT_CLEAR
- CPMAC_MAC_EOI_VECTOR – End Of Interrupt

Driver uses these to know:

- when a packet is sent
- when a packet is received

◇ 7. DMA Pointer Registers

These control the **DMA descriptor rings**.

For TX:

- CPMAC_TX_PTR(channel) – current TX descriptor pointer
- CPMAC_TX_ACK(channel) – TX completion ack

For RX:

- CPMAC_RX_PTR(channel) – RX descriptor pointer
- CPMAC_RX_ACK(channel) – RX completion acknowledgment

These tell MAC:

- where the buffer descriptors are located
- when a buffer is consumed
- when a buffer is produced
- signal RX/TX completion to driver

◇ 8. STATISTICS REGISTERS

From CPMAC_STATS_RX_GOOD to CPMAC_STATS_TX_OCTETS

These count:

- good frames
- broadcast/multicast frames
- CRC errors
- collisions
- overruns
- jabbers
- bytes sent/received
- flow control events

Driver can use these for **ethtool -s stats**.

◇ Summary in simple terms:

Register group	Purpose
TX/RX Control	Enable/disable TX & RX engines, teardown DMA
MBP Filtering	Control what packets MAC accepts
MAC Address Registers	Store MAC address & multicast filter
MAC Control	Set full-duplex, MII/RMII, flow control, loopback
Interrupt Registers	Enable/clear TX/RX/MAC IRQ
DMA Pointers	Point to descriptor rings
Statistics	Counters for packets, errors, collisions

Below is a **clean, theoretical, step-by-step explanation of the RX**

(receive) path of the CPMAC driver you posted — **completely without code**, exactly as you prefer.

CPMAC RX PATH — FULL EXPLANATION (THEORY ONLY)

Think of the RX path as a **pipeline**:

4. **Hardware DMA engine**
5. **Descriptor ring**
6. **NAPI poll**
7. **SKB creation**
8. **Packet delivery to the network stack**

Now let's go deep into each part.

1. RX DESCRIPTOR RING – THE HEART OF RECEIVE

The driver maintains a **ring (circular list)** of descriptors.

Each descriptor contains:

- pointer to buffer (SKB data area)
- buffer length
- flags (OWN, SOP, EOP, EOQ...)
- DMA physical address

Meaning of important flags:

Flag	Meaning
OWN	Hardware owns the buffer. Driver must not touch it.
SOP	Start of Packet
EOP	End of Packet
EOQ	End-of-Queue — hardware reached end of the ring.
datalen	Number of bytes received.
buflen	Size of allocated buffer

The hardware fills a descriptor only if **OWN = 1** (owned by HW).

2. HARDWARE RX OPERATION

Hardware behaviour:

9. HW reads RX_PTR register → points to a descriptor.
10. If descriptor has **OWN set**:
 - HW places received packet into buffer
 - HW clears OWN (meaning “driver now owns it”)
 - HW fills datalen
11. If hardware reaches EOQ:
 - It stops
 - Driver must restart RX by writing RX_PTR again.

Key concept:

Hardware does **not** automatically wrap the ring.

The driver must maintain circular linking.

3. INTERRUPT ARRIVES → NAPI STARTS

When RX interrupt comes:

- Interrupt handler disables further RX interrupts
- Enables NAPI poll scheduling
- NAPI poll function (cpmac_poll()) is executed

Why NAPI?

To avoid interrupt storms for high traffic.

4. NAPI RX POLL — THE MAIN RX LOGIC

Inside cpmac_poll():

(A) Lock the RX ring

Because descriptors can be modified in parallel by hardware.

(B) Start from rx_head

The descriptor that we expect to hold new data.

(C) Check if OWN = 0

OWN = 0 ⇒ **hardware has completed writing a packet.**

If OWN = 1 ⇒ no more packets; poll ends.

5. PROCESS ONE PACKET

For each descriptor:

12. Call cpmac_rx_one()

This function performs:

- Unmap DMA (DMA_FROM_DEVICE)
- Set SKB length = datalen
- Set protocol = eth_type_trans()
- Update RX statistics
- Return SKB to upper layers

13. Send SKB to network stack

netif_receive_skb()

14. Allocate a new SKB

(For reusing this descriptor)

15. Map new SKB for DMA

DMA engine can again fill this buffer.

16. Set OWN back to hardware

6. HANDLING EOQ (END OF QUEUE)

Why EOQ happens?

Hardware hits the end of ring and stops because the “next” pointer was not yet written (timing race).

If driver sees EOQ flag:

- Remember this descriptor as restart
- After freeing some buffers, restart the RX DMA by:
Write RX_PTR with restart->mapping

This reactivates hardware receive operation.

7. WHEN PACKET BUDGET IS REACHED

NAPI must process up to budget packets only.

If budget reached:

- poll returns 1 (meaning more work to do)

- NAPI will call poll again

8. WHEN NO MORE PACKETS

If no more packets:

17. Driver completes NAPI:

```
napi_complete()
```

18. Re-enable RX interrupts:

```
write CPMAC_RX_INT_ENABLE = 1
```

This allows next interrupt to wake NAPI again.

9. RING OVERFLOW / MISMATCH HANDLING

If:

- driver sees EOQ but OWN == 0 (illegal state)
- or any descriptor looks inconsistent

This is considered **fatal error**.

Then driver:

19. Stops RX
20. Disables NAPI
21. Schedules hardware reset
22. Dumps all descriptors
23. Resets CPMAC hardware

This avoids permanent RX lock-up.

10. UPDATING rx_head

After processing all descriptors:

`rx_head = next descriptor to be checked`

This is the descriptor hardware will fill next time.

SUMMARY — WHAT HAPPENS IN RX PATH

Step-by-step pipeline flow:

24. **Hardware fills descriptor, clears OWN**
25. **Interrupt arrives**
26. **NAPI starts**
27. For each descriptor:
 - If OWN==0 → packet available
 - Unmap DMA
 - Process SKB
 - Pass to stack
 - Allocate new SKB
 - Map for DMA
 - Set OWN = HW owns again
28. **Move to next descriptor**
29. If EOQ encountered → restart DMA
30. If no more packets → stop NAPI + enable interrupts
31. If ring corrupt → restart hardware

If you want, I can also explain:

- ✓ TX path
- ✓ DMA mapping
- ✓ Descriptor life cycle diagram
- ✓ What exactly to print to debug RX
- ✓ How to modify RX path
- ✓ How to optimize ring size
- ✓ How to add your own printk() for learning

Just tell me:

"Explain TX path", "Explain interrupt flow", "Explain where to add my prints", etc.

Below is the **clearest and simplest deep-theoretical explanation** of the difference between **MII** and **MDIO**, because many people confuse these two.

★ SHORT ANSWER (very clear)

Term	Full Form	Purpose	Used For
MII	Media Independent Interface	Carries actual Ethernet data between MAC ↔ PHY	Sending & receiving packets
MDIO	Management Data Input/Output	Manages/configures the PHY	Reading PHY registers, link status, speed, duplex

MII = Data Path

MDIO = Control Path

These two are completely different and serve different purposes.

★ DETAILED EXPLANATION

◇ 1. What is MII? (Media Independent Interface)

Purpose

MII is the **physical data interface** between:

- MAC (Ethernet controller inside SoC)
- PHY (external or internal Ethernet transceiver)

MII transfers actual Ethernet frames:

- TX data
- RX data
- Clocks
- Control signals

Signals inside MII

MII is a **parallel bus** with many signals:

TX (MAC → PHY):

- TXD0-TXD3 (4 bits)
- TX_EN (transmit enable)
- TX_CLK (clock)

RX (PHY → MAC):

- RXD0-RXD3 (4 bits)
- RX_DV (data valid)

- RX_CLK (clock)
- CRS, COL (carrier sense, collision)

Variants

Interface	Speed	Pins
MII	10/100 Mbps	16–18 pins
RMII	10/100 Mbps (reduced pins)	~8 pins
GMII	1000 Mbps	More pins
RGMII	1000 Mbps (reduced pins)	~12 pins

◊ 2. What is MDIO? (Management Data I/O)

Purpose

MDIO is **not for data**.

MDIO is used by the MAC to **control and query** the PHY.

This includes:

- Reading link status
- Setting speed (10/100/1000)
- Setting duplex
- Resetting PHY
- Reading vendor-specific registers

Signals in MDIO

Only **two wires**:

- **MDC** – Management Data Clock
- **MDIO** – Bi-directional data line

How many PHYs can it control?

MDIO can control **up to 32 PHYs** (address 0–31).

MAC sends commands like:

read register X from PHY Y

write register Z to PHY Y

★ Example: AM335x / BeagleBone Black

- The AM335x has a **CPSW (Ethernet MAC)**.

- It has MDIO module to control PHYs.
- It uses **RGMII** as data interface (variant of MII).
- MDIO is used to:
- discover PHY addresses
- detect link up/down
- set speed/duplex
- check auto-negotiation

★ Why do drivers use MDIO but not MII?

Because:

MII = hardware electrical signals

You cannot manipulate them in software.

MDIO = software-accessible register bus

Kernel drivers use MDIO operations:

- mdiobus_read()
- mdiobus_write()
- PHY state machine
- phy_start()
- phy_connect()

★ SUPER SIMPLE UNDERSTANDING

MII = Highway

Where **actual cars (Ethernet data)** flow between MAC and PHY.

MDIO = Traffic control system

Used to:

- read traffic lights
- change speed limits
- control the highway
- check if lane is open

They are different systems.

★ FINAL SUMMARY

Feature	MII	MDIO
What it does	Sends/receives Ethernet frames	Configures the PHY
Type	Data interface	Management interface
Wires	Many (8–18+)	2
Speed	Carries actual line speed	Very slow (2.5 MHz clock)
Who uses it?	Hardware (MAC \leftrightarrow PHY)	Software (driver \leftrightarrow PHY)

If you want, I can explain:

- ✓ Difference between MII, RMII, GMII, RGMII
- ✓ How Linux detects PHY over MDIO
- ✓ How `phy_connect()` uses MDIO internally
- ✓ How auto-negotiation works over MDIO

Just tell me!

concepts

Tuesday, November 25, 2025 5:21 PM

Ah, this is a common point of confusion. The key is understanding **DMA (Direct Memory Access)**. In real NICs, the CPU is **mostly out of the RX/TX data path**. Let me explain step by step.

1. NIC uses DMA for RX

- Modern NICs have a **DMA engine**.
- When you set up the RX ring, you give the NIC **physical addresses of memory buffers**.
- NIC writes incoming packets **directly into these memory buffers**, without CPU intervention.
- CPU only touches the descriptors to **check which packets are ready** or to **reclaim buffers**.

So even if CPU is busy in interrupts, the NIC can still store packets in memory.

2. Why interrupts are disabled

- Interrupts are disabled **only for RX interrupts**, not for CPU execution.
- The NIC keeps receiving packets via DMA into the RX ring.
- Once the CPU gets scheduled to run poll(), it reads the descriptors **already filled by NIC**.

3. Polling vs interrupt handling

- **Without NAPI:** Each packet raises an interrupt → CPU handles every packet individually → high CPU overhead.
- **With NAPI:** First interrupt disables further interrupts → CPU polls **multiple packets at once** → much more efficient.
- CPU doesn't need to handle every packet immediately; it can process a **batch**.

4. Summary

Component	Role
NIC	Receives packets, writes to RX descriptors using DMA
CPU ISR	Initially triggered on first packet, disables further interrupts, schedules NAPI poll
NAPI poll	Processes all ready RX descriptors (packets) in batches
Interrupts re-enabled	If not all descriptors are full, NIC can trigger new interrupts

Key insight: CPU is **not used to copy packets from NIC to memory** — that's why RX can continue even if CPU is busy.

Okay — you want the explanation **as a real-world scenario** of how actual Ethernet packets move **from cable → PHY → MAC → DMA → driver → kernel, exactly matching your probe() flow**.

No code.

Only a realistic **live Ethernet example** as if you can “see packets traveling”.

Let's do it step-by-step.

REAL LIFE ETHERNET SCENARIO

(Same as how your cpmac_probe() driver will work on the BeagleBone)

To make it easy, imagine this network:

Laptop <—→ BeagleBone Black (BBB)

Laptop IP: 192.168.1.10

BBB IP: 192.168.1.20

Laptop sends a ping to BBB.

We will walk through EXACTLY what happens from wire → PHY → MAC → DMA → driver → kernel → userspace.

Before any packet arrives: What probe() already did

cpmac_probe() completed these setups:

✓ MAC registers mapped

Driver knows:

- where the TX/RX DMA registers are
- where control registers are
- ✓ IRQ assigned

Driver knows:

- which interrupt line Ethernet uses
- ✓ PHY found

PHY lives on MDIO bus, e.g.

- bus: 0
- phy_id: 1
- ✓ PHY connected

Driver + PHY framework handle:

- cable detect
- link up/down
- speed (100Mbps)
- duplex
- ✓ NAPI registered

Interface is ready to receive and send packets.

NOW THE REAL MAGIC STARTS

Laptop sends ping.

Let's watch the packet like a movie.

1. Laptop puts an Ethernet frame on wire

Laptop sends a real electrical signal at 100 Mbps:

[Destination MAC = BBB_MAC]

[Source MAC = LAPTOP_MAC]

[Type = IPv4]

[Payload = ICMP ping request]

[FCS]

This is NOT yet bytes — it is **voltage transitions on the twisted pair cable**.

2. PHY on BeagleBone receives the signal

BBB's PHY (DP83867 on some boards):

- recovers the clock
- decodes MLT-3 modulation
- extracts raw bitstream
- reconstructs Ethernet frame

So now the PHY has the full Ethernet frame in its internal FIFO.

3. PHY forwards packet to MAC (using MII/RMII)

Depending on BBB board:

- RMII (2-bit interface, 50MHz clock)
- or MII (4-bit, 25MHz clock)

PHY sends:

- frame start
- bytes of the packet
- frame end

These bytes are clocked into the **CPSW / CPMAC hardware MAC block**.

MAC now has the entire frame.

4. MAC pushes packet into RX DMA ring

When probe() ran, driver set ring_size = 64.

So MAC has **64 RX descriptors** in RAM.

Each descriptor represents:

[pointer to empty packet buffer in RAM]

[length]

[flags]

MAC picks the next free descriptor:

RX desc #5

And writes the received packet into the buffer pointed by desc #5.

Then MAC marks:

OWN bit = CPU owns buffer now

RX_DONE = 1

5. MAC triggers an interrupt

RX complete interrupt fires:

IRQ: ethernet-rx

CPU wakes the interrupt handler in the driver.

6. Interrupt handler does NOT process packets

(because of NAPI)

Interrupt handler merely says:

Schedule NAPI poll

Disable further RX interrupts

Why?

Because packet storms would overwhelm CPU if processed in interrupt context.

7. cpmac_poll() runs (NAPI polling)

This is where Linux actually processes packets.

cpmac_poll():

Step 1 → Looks at RX descriptor #5

It sees:

RX_DONE = 1

OWN = CPU

Meaning:

“A new packet is ready for me”

Step 2 → Allocates an skb

It copies packet from DMA buffer into a kernel SKB.

Step 3 → Hands it to networking stack

`netif_receive_skb()`

Linux networking then:

- checks VLAN
- checks routing
- checks firewall
- sends to IP layer
- sends to ICMP layer
- sends to socket layer

Step 4 → Refill the RX descriptor

Buffer reused for the next packet.

💡 8. Kernel delivers packet to ping process

ICMP process running in userspace receives:

ICMP echo request

and sends back:

ICMP echo reply

🚀 9. Now the TX path

To send reply, Linux eventually calls:

`cptmac_start_xmit()`

This function:

- ✓ chooses a free TX descriptor
- ✓ copies packet into DMA TX buffer
- ✓ tells MAC: "Transmit this"
- ✓ MAC clocks bytes to PHY (via RMII/MII)
- ✓ PHY puts electrical signals onto cable

→ back to laptop

🌟 10. TX completes, interrupt fires

MAC issues TX complete interrupt:

Driver frees TX buffer

Updates stats

Re-enables interrupts (if NAPI)

Packet is gone out.

🌟 FINAL RESULT

You now see:

Reply from 192.168.1.20: bytes=64 time=1ms

Laptop successfully pinged the BBB.

💧 THIS IS EXACTLY HOW YOUR DRIVER HANDLES PACKETS

Your probe() just prepared all of this:

What probe set up	Why it's needed
DMA ring buffers	Where PHY/MAC place packets
IRQ	When packet arrives
net_device ops	How Linux asks driver to send packet
PHY connection	Manage link speed, duplex
MAC registers	Control transmitting and receiving
NAPI	Efficient packet handling

After that →

actual Ethernet hardware + DMA + PHY + MAC take over the real data movement.

Let's break this into **very simple, real-world explanations**.

This part of the CPSW probe() confuses many people because it looks mysterious — but it's actually simple if you understand **Runtime PM (Power Management)**.

Why these lines exist? (Simple Summary)

These lines handle **turning ON the CPSW hardware block** so that the driver can safely access registers.

CPSW hardware block = inside AM335x SoC

It gets **powered OFF** by Linux when not used (Runtime PM).

So the driver must:

1. **Enable Runtime PM**
2. **Power ON the CPSW module**
3. **Keep it powered as long as driver is using it**

Now let's explain each line with an easy real-world scenario

Imagine CPSW is a *room with lights off*.

If you want to enter and work inside (read/write registers), you must:

- Turn ON the light → pm_runtime_get_sync()

- When you leave, turn OFF the light → pm_runtime_put()

1. pm_runtime_enable(dev)

```
pm_runtime_enable(dev);
```

Meaning

This tells the Linux kernel:

“This device will use Runtime Power Management. Manage its power states automatically.”

Real-world example

Think of this as telling your apartment building:

“I want automatic lighting and electric control for my room.”

Nothing is powered ON yet — you just **enable the capability**.

Why needed for CPSW?

The CPSW (Ethernet controller) can be:

- Powered ON when network is active
- Powered OFF when unused to save power

So the driver must explicitly enable power management.

2. pm_runtime_get_sync(dev)

```
ret = pm_runtime_get_sync(dev);
```

Meaning

This function:

- ✓ Powers ON the hardware block
- ✓ Enables the clock (fck)
- ✓ Makes register access safe
- ✓ Waits synchronously until power ON completes

Real-world example

This is like:

“Turn ON the lights in my room so I can work.”

If lights are already ON

Nothing happens (reference count increases).

If lights are OFF

Kernel powers the CPSW module → enables clocks → resets the hardware → then returns.

What if turning ON fails?

```
if (ret < 0) {
    pm_runtime_put_noidle(dev);
    goto clean_runtime_disable_ret;
```

}

Meaning:

- If hardware failed to power ON
- Or clock failed
- Or something is wrong with SoC PM firmware

Then the driver cleans up and exits.

💡 Why is this important in CPSW?

Because the driver needs to access registers:

`cpsw->regs = devm_ioremap_resource()`

`cpsw->wr_regs = devm_platform_ioremap_resource()`

and initialize things like:

- ALE (Address Lookup Engine)
- CPTS (timestamping)
- Slaves / ports
- Switch subsystem registers

If the hardware is still in **low-power mode**, reading registers will crash the kernel.

💡 Putting it all together

Code	Meaning
<code>pm_runtime_enable(dev);</code>	Tell kernel "This device supports dynamic power management."
<code>pm_runtime_get_sync(dev);</code>	Actually power ON the CPSW hardware block.
<code>pm_runtime_put_noidle(dev);</code>	Undo power ON request if things go wrong.

💡 Small Diagram

Driver loads



Enable Runtime PM



Turn on CPSW clocks & power domain



Now driver can safely access CPSW registers



Continue probe() initialization

Let's explain exactly what this line does, what `cpsw_probe_dt()` is, why it is called here, and what happens if it fails — specifically in context of the **CPSW (TI Ethernet Switch) driver**.

What is `cpsw_probe_dt()`?

It means:

Parse the Device Tree (DT) and fill `cpsw->data` with all hardware configuration information of CPSW.

DT = Device Tree, which is where the SoC describes:

- How many ports?
- What PHY is connected?
- What MAC addresses?
- What interrupt mapping?
- What is the switch mode?
- What kind of CPSW version (CP(M)AC / CPSW2G / CPSW3G)?
- DMA channels?
- ALE settings?
- Internal or external PHY?
- Pinmux?
- Bus frequencies?
- VLAN / dual-emac mode / switch mode?

The CPSW driver **cannot function** without this information.

So `cpsw_probe_dt()` extracts all this from the DT and puts it inside:

`cpsw->data // struct cpsw_platform_data`

Full meaning of the line

```
ret = cpsw_probe_dt(&cpsw->data, pdev);
```

It does:

4. Search for the node "cpsw" inside the device tree.
5. Read properties of that node.
6. Store them into `cpsw->data`.

Why does the driver need `cpsw->data`?

Because CPSW can behave in different modes:

Example scenarios

Scenario 1: BeagleBone Black (Dual EMAC mode)

The board has:

- 2 external PHYs (RGMII)
- 2 MAC addresses
- No switch forwarding (dual-emac mode)

The DT tells the driver:

```
slaves = 2
slave[0].phy_id = 0
slave[1].phy_id = 1
mode = dual_emac
```

Scenario 2: Switch Mode (CPSW as 3-port switch)

Some SoCs run CPSW in switch mode:

```
slaves = 2 ports
ale_enabled = yes
switch_mode = yes
```

Scenario 3: Internal PHY or no PHY

Some boards have *internal PHY*:

```
slave[0].phy-handle = <&internal_phy>;
```

Others use *MII*, *RMII*, *RGMII* modes, each described in DT.

🔍 What exactly is inside cpsw->data?

struct cpsw_platform_data contains critical configuration:

- number of N/W slave ports
- PHY interface type (MII/RMII/RGMII)
- PHY addresses
- VLAN config
- CPTS (timestamping) enabled?
- ALE (switch engine) enabled?
- dual_emac mode?
- GMII_SEL settings
- hardware version info
- quirks or SoC-specific flags

Without these, CPSW cannot communicate with PHY.

❗ Why check return value?

```
if (ret)
    goto clean_dt_ret;
```

Because if DT parsing fails:

- Driver does **NOT** know number of ports
- Driver does **NOT** know PHY address
- Driver does **NOT** know interface mode
- Driver might write to **wrong registers**
- Device may not exist / misconfigured

So it must abort probe.

Simple Real-Life Example

Imagine CPSW is a 3-port Ethernet switch inside the SoC, and DT describes it like:

```
cpsw {  
    slaves = 2;  
    slave0-phy = PHY at MDIO address 2  
    slave1-phy = PHY at MDIO address 3  
    mode = "dual_emac";  
}
```

cpsw_probe_dt() reads these values and fills:

```
cpsw->data.slaves = 2  
cpsw->data.slave[0].phy_id = 2  
cpsw->data.slave[1].phy_id = 3  
cpsw->data.dual_emac = 1
```

Without this information, CPSW has **no idea where to look for PHY** or how many ports exist.

What does clean_dt_ret do?

If DT parsing fails, driver must:

- power-down device (runtime_pm_put / disable clocks)
- free allocated memory
- stop probe sequence

So it jumps to cleanup and exits.

TL;DR — Easy Version

cpsw_probe_dt() = Read everything about CPSW from the device tree and configure driver.

If DT parsing fails → driver cannot continue → abort.

Great — **this is one of the MOST important structs in the CPSW driver**, and understanding it will unlock the entire driver flow.

I will explain:

- What **struct cpsw_platform_data** contains
- Why **each field** exists
- What **role each plays** in running the CPSW Ethernet MAC
- Real-world examples from BeagleBone Black
- How the driver uses each field

Everything in a **clear, beginner-friendly** way.

Probe dt function in detail

★ FIRST — What is struct cpsw_platform_data?

It is a **configuration container** for CPSW.

This struct tells the driver **what kind of CPSW hardware we have** and instructions on how to use it.

Linux fills this struct using:

cpsw_probe_dt()

The struct is defined in:

drivers/net/ethernet/ti/cpsw.h

★ FULL STRUCT (for reference)

Here is the structure (simplified):

```
struct cpsw_platform_data {  
    u32 slaves;  
    u32 active_slave;  
    u32 cpsw_version;  
  
    struct cpsw_slave_data *slave_data;  
    u32 ale_entries;  
    u32 bd_ram_size;  
  
    bool dual_emac;  
    bool switch_mode;  
    bool cptp_enabled;  
  
    u8 mac_addr[ETH_ALEN];  
    u32 host_port_reg_offset;  
  
    struct cpsw_ale_params ale_params;  
};
```

Let's explain every field **with examples**.

🔍 DETAILED EXPLANATION OF EACH FIELD

1. u32 slaves;

Meaning:

Number of slave ports available on CPSW.

In CPSW:

- CPSW has **2 ports** that connect to PHYs
- Plus **1 host port** (connection to Linux networking stack)

Example (BeagleBone Black):

`slaves = 2;`

Used by driver to create:

- `net_device` for `eth0`
- `net_device` for `eth1`

2. u32 active_slave;

Meaning:

Which slave is “active” when in *single* EMAC mode.

If only one Ethernet interface is used, which PHY port?

Example:

If only port 1 is enabled:

`active_slave = 0 or 1`

Often unused in dual-emac mode.

3. u32 csw_version;

Meaning:

Hardware version:

- CPSW2G (AM335x)
- CPSW3G (AM437x)
- CPSW9G (AM57xx)

The driver uses this to apply quirks.

Example:

`cpsw_version = CPSW_VERSION_2;`

Because AM335x = CPSW2G.

4. struct csw_slave_data *slave_data;

This is an **array**, one entry per slave port.

slave_data contains:

- PHY address
- PHY connection type (MII / RMII / RGMII)
- Port VLAN config
- GMII selection register offset

Example:

For BeagleBone Black:

```
slave_data[0].phy_id = 0;  
slave_data[1].phy_id = 1;
```

These are MDIO bus addresses.

The driver uses this to call:

```
of_phy_connect()
```

5. u32 ale_entries;

ALE = Address Lookup Engine (switch table).

This field tells driver how many entries held by the switch hardware.

Example:

```
ale_entries = 1024;
```

If switch mode enabled, the driver uses this to:

- create forwarding table
- program MAC address filtering
- VLAN filtering

6. u32 bd_ram_size;

Buffer Descriptor RAM size in CPSW.

CPSW has internal SRAM used by TX/RX DMA.

Example:

```
bd_ram_size = 0x2000;
```

Used by the DMA driver to allocate descriptors.

7. bool dual_emac;

Meaning:

Enable “dual EMAC mode” = each slave port becomes **its own network interface**.

This is how BeagleBone Black works:

eth0 = slave 0

eth1 = slave 1

Why needed?

Because CPSW normally is a *switch* (bridged mode).

Dual-emac turns it into 2 independent MACs.

8. bool switch_mode;

Enable 3-port hardware switch behavior:

eth0 \leftrightarrow switch \leftrightarrow eth1

Used in industrial applications where ports forward frames internally.

9. bool cpts_enabled;

Enables CPTS = Common Platform Time Sync (IEEE-1588 clock).

Used for:

- PTP time stamping
- Ethernet time sync

Example:

cpts_enabled = 1;

10. u8 mac_addr[ETH_ALEN];

MAC address for the host port (HW port 0).

Driver copies this to net_device.

If not present, driver generates a random one.

11. u32 host_port_reg_offset;

Offset to access host port registers inside switch subsystem.

Different SoC revisions place registers differently.

How cpsw_platform_data is used inside probe()

Step 1 — CPSW maps registers

Step 2 — Probe DT

Fills this struct.

Step 3 — Driver configures CPSW hardware:

- reset CPSW
- configure ALE
- configure CPTS
- configure DMA
- configure GMII_SEL
- configure RX/TX channels

Step 4 — Create net devices using data from this struct:

```
cpsw_create_ports();  
register_netdev();
```

★ Real-world Example (BeagleBone Black DT)

DT snippet:

```
ethernet@4a100000 {  
    slaves = <2>;  
    dual_emac = <1>;  
    slave1: ethernet-phy@0 { ... };  
    slave2: ethernet-phy@1 { ... };  
    cpts-en = <1>;  
};
```

Becomes:

```
cpsw->data.slaves = 2  
cpsw->data.dual_emac = true  
cpsw->data.cpts_enabled = true  
slave_data[0].phy_id = 0  
slave_data[1].phy_id = 1
```

⌚ TL;DR — SIMPLE VERSION

cpsw_platform_data describes:

- How many ports CPSW has
- What PHYs are connected
- Mode: dual-emac or switch
- What hardware features exist (CPTS, ALE, versions)
- RAM size
- MAC address

The driver uses this struct to **configure the CPSW hardware and create the network interfaces.**

Okay — now we're stepping into the **Device Tree (DT) parsing part** of the CPSW probe function.

You asked specifically about:

```
struct device_node *node = pdev->dev.of_node;  
struct device_node *slave_node;  
x
```

Let me explain **what they are, why they are needed, and exactly what they do for the CPSW (TI Ethernet Switch) driver**, using clear, real-world examples.

★ 1. What is struct device_node *node?

This is a pointer to the **Device Tree node** representing the CPSW hardware.

In simple words:

node points to the section of the device tree where CPSW is described.

Device tree example (BeagleBone Black):

```
ethernet@4a100000 {  
    compatible = "ti,cpsw";  
    slaves = <2>;  
    dual_emac = <1>;  
    ...  
  
    slave0 {  
        phy_id = <&davinci_mdio 0>;  
        phy-mode = "rmii";  
    };  
  
    slave1 {  
        phy_id = <&davinci_mdio 1>;  
        phy-mode = "rmii";  
    };  
};
```

When the kernel matches "ti,cpsw" with this node, the platform device gets created.

Then:

```
struct device_node *node = pdev->dev.of_node;
```

node now points to the **ethernet@4a100000** node.

★ Why CPSW uses this node?

Because the driver must read:

- How many slave ports
- What PHY each slave uses
- PHY mode (MII/RMII/RGMII)
- Switch mode or dual-emac mode

- MAC addresses
- CPTS enable flag
- ALE table size
- Child nodes for slaves

So this one pointer (node) gives access to EVERYTHING.

★ 2. What is struct device_node *slave_node?

This is used to iterate through **child nodes** inside the CPSW node.

Example DT:

```
ethernet@4a100000 {
    slave0 {
        phy-handle = <&phy0>;
        phy-mode = "rmii";
    };
}
```

```
slave1 {
    phy-handle = <&phy1>;
    phy-mode = "rmii";
};
};
```

`slave_node` will point to these child nodes:

`slave_node → "slave0"`
`slave_node → "slave1"`

CPSW has **2 network interface ports**, so:

The driver must read DT for each slave:

- PHY address
- PHY mode
- VLAN tagging
- MAC address override
- MDIO handle

This is done like:

```
for_each_child_of_node(node, slave_node) {
    parse slave0
    parse slave1
}
```

★ Why CPSW MUST parse slave nodes?

Because:

CPSW concept	DT child node
Slave port 0	slave0 DT node
Slave port 1	slave1 DT node

Each port has **its own PHY** with its own MDIO address.

Example:

slave0.phy_id = 0

slave1.phy_id = 1

If the driver does NOT parse these nodes →

it will NOT know which PHY belongs to which port → network cannot come up.

★ Real-world Example — Let's imagine the driver doing this

DT:

```
ethernet@4a100000 {
```

```
    slaves = <2>;
```

```
    slave0 {
```

```
        phy-handle = <&phy0>;
```

```
        phy-mode = "rmii";
```

```
    };
```

```
    slave1 {
```

```
        phy-handle = <&phy1>;
```

```
        phy-mode = "rmii";
```

```
    };
```

```
};
```

Driver:

```
node = pdev->dev.of_node; // → points to ethernet@4a100000
```

for child in node:

```
    slave_node = "slave0" // first iteration
```

```
    parse phy, mode, etc.
```

```
    slave_node = "slave1" // second iteration
```

```
    parse phy, mode, etc.
```

★ How the driver uses slave_node internally?

Example:

```
phy_node = of_parse_phandle(slave_node, "phy-handle", 0);
phy_id = of_mdio_parse_addr(node, phy_node)
```

and

```
of_get_phy_mode(slave_node)
```

This fills:

```
cpsw->data.slave_data[0]
cpsw->data.slave_data[1]
```

Used later to create network interfaces.

★ TL;DR — Simple Version

node

- Points to the CPSW device tree node
 - Contains global CPSW configuration
 - Required for parsing slaves, ALE, CPTS, mode, etc.
- slave_node
- Points to child nodes inside CPSW node
 - Each child represents a CPSW Ethernet port
 - Used to get PHY address, PHY mode, etc.

Let's break this down **like a beginner**, slowly and clearly.

You asked:

What is happening in this code? What is "slaves" property? Why read it? What is data->slaves?

1. The exact code we are understanding

```
if (of_property_read_u32(node, "slaves", &prop)) {
    dev_err(&pdev->dev, "Missing slaves property in the DT.\n");
    return -EINVAL;
}
data->slaves = prop;
```

2. What is being read? → A DeviceTree property

node is the DT node for the CPSW MAC:

```
/* node = pdev->dev.of_node */
```

The code is trying to read:

```
slaves = <2>;
```

from the DeviceTree:

Example CPSW node (BeagleBone Black):

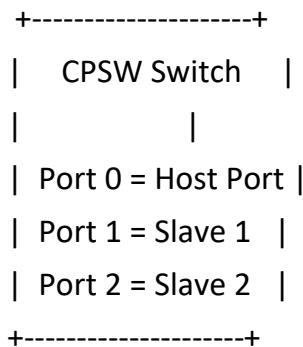
```
cpsw: ethernet@4a100000 {  
    slaves = <2>; // CPSW has 2 slave ports (port 1 and port 2)  
    active_slave = <0>;  
    ...  
}
```

⚡ This “slaves” property tells the driver:

How many MAC ports does the hardware have?

3. Why is "slaves" needed?

TI CPSW (BeagleBone Black Ethernet MAC) architecture:



So total "slaves" = **2** (Port 1 and Port 2).

The driver **must know how many ports to initialize**:

- How many network interfaces to create (eth0, eth1)
- How many PHYs to attach
- How many RX/TX queues to prepare
- How many NAPI structures to create
- How many IRQs to request

If your hardware has only 1 port → slaves = 1

If 2 ports → slaves = 2

4. Step-by-step explanation of the code

Step 1: Try reading DeviceTree property "slaves"

```
if (of_property_read_u32(node, "slaves", &prop)) {
```

- of_property_read_u32 returns 0 on success
- non-zero (true) on failure

So if "slaves" is missing → error.

 This catches DT mistakes.

Step 2: Error printed

```
dev_err(&pdev->dev, "Missing slaves property in the DT.\n");
```

```
return -EINVAL;
```

Driver cannot continue → exits probe.

Step 3: Save slaves count

```
data->slaves = prop;
```

Now driver knows how many CPSW MAC ports exist.

5. What is data->slaves?

It is a field of:

```
struct cpsw_platform_data {  
    int slaves; // number of slave ports  
    ...  
}
```

Let's break this code **slowly and clearly** so you understand EXACTLY what happens and WHY.

The code

```
data->slave_data = devm_kcalloc(&pdev->dev,  
                                 data->slaves,  
                                 sizeof(struct cpsw_slave_data),  
                                 GFP_KERNEL);  
  
if (!data->slave_data)  
    return -ENOMEM;
```

1. What is being allocated?

The driver allocates an **array** of:

```
struct csw_slave_data
```

One entry **per CPSW slave port**.

You already saw:

```
data->slaves = 2;
```

So if slaves = 2:

Allocate array of 2 x struct csw_slave_data

This looks like:

Index 0 → slave port 1

Index 1 → slave port 2

2. What is devm_kcalloc()?

It means:

- **Allocate memory** (like kcalloc)
- **Zero-initialize it** (calloc)
- Use **devm_** → the memory will be *automatically freed* when driver is removed

[Why devm?](#)

Because:

- When driver probe **fails** → memory auto-freed
- When driver is unloaded (rmmod) → auto-freed
- You do NOT need kfree()

This prevents memory leaks.

3. Meaning of parameters

```
devm_kcalloc(&pdev->dev, // device that owns memory
```

```
    data->slaves, // number of elements
```

```
    sizeof(struct csw_slave_data), // size of each element
```

```
    GFP_KERNEL); // normal kernel allocation flag
```

So memory size =

```
data->slaves * sizeof(struct csw_slave_data)
```

If slaves = 2 and struct is 64 bytes:

Alloc = 2 * 64 = 128 bytes

Zero-filled memory returned.

4. What is data->slave_data?

It becomes a pointer to the array:

data->slave_data[0] → info for slave port 1 (eth0)
data->slave_data[1] → info for slave port 2 (eth1)

Later, driver fills this data from DeviceTree.

5. What is stored inside struct cpsw_slave_data?

Something like:

```
struct cpsw_slave_data {  
    u8 mac_addr[ETH_ALEN];  
    int phy_id;  
    char phy_id_name[32];  
    int slave_port;  
    struct device_node *phy_node;  
    struct device_node *slave_node;  
    struct cpsw_slave_platform_data *slave_data;  
};
```

Important fields:

`mac_addr`

- From DT
 - MAC for that specific port
- `phy_node`
- DT node for PHY connected to that port
(e.g., MDIO address)
- `slave_node`
- The slave@0 or slave@1 node
- `phy_id`
- PHY address on MDIO bus

6. Why must we have **one structure per slave port?**

Because each CPSW port has independent:

- PHY
- MAC address
- TX/RX queue mapping
- IRQ mapping

- NAPI context
- Port VLAN ID
- Flow control settings

So the driver needs **per-port configuration**.

Example:

`eth0 → uses slave port 1 → MDIO phy = 0`

`eth1 → uses slave port 2 → MDIO phy = 1`

So:

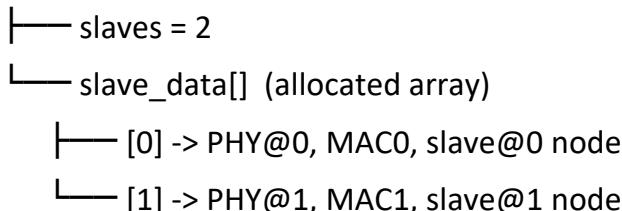
`data->slave_data[0].phy_id = 0;`

`data->slave_data[1].phy_id = 1;`

Without per-port struct → driver cannot support multiple MAC ports.

Simple visual diagram

`cpsw->data` contains platform-level info



Summary

Step	Action
Read DT slaves property	<code>data->slaves = 2</code>
Allocate per-slave array	<code>data->slave_data</code>
Array contains	struct <code>cpsw_slave_data</code> for each port
Why <code>devm_kcalloc</code>	Memory auto-managed
Why needed	Every port has own PHY, MAC, IRQ, NAPI

Let's break this line **very clearly and practically**, because it confuses many beginners:

The code

```
ret = of_platform_populate(node, NULL, NULL, &pdev->dev);

/* We do not want to force this, as in some cases may not have child */
if (ret)
    dev_warn(&pdev->dev, "Doesn't have any child node\n");
```

1. What is of_platform_populate()?

It is a DeviceTree helper that:

Creates platform devices for all child nodes of a given DT node.

For example, if your DeviceTree has:

```
cpsw@4a100000 {
    slaves = <2>

    slave@0 {
        phy-handle = <&phy0>;
    };

    slave@1 {
        phy-handle = <&phy1>;
    };

    mdio@4a101000 {
        #address-cells = <1>;
        #size-cells = <0>;

        phy0: ethernet-phy@0 { ... };
        phy1: ethernet-phy@1 { ... };
    };
}
```

Then

```
of_platform_populate(node, NULL, NULL, &pdev->dev);
```

creates platform devices for:

```
cpsw/slave@0 (child)
```

```
cpsw/slave@1 (child)
cpsw/mdio@4a101000 (child)
cpsw/mdio@4a101000/ethernet-phy@0 (grandchild)
cpsw/mdio@4a101000/ethernet-phy@1 (grandchild)
```

The kernel internally calls:

```
platform_device_register()
```

for each child node.

💡 2. What this means for CPSW driver?

It means:

The CPSW driver does NOT manually create PHY or MDIO devices.

DT takes care of it.

🔍 3. Why do we need child devices for CPSW?

Because CPSW has internal subsystems:

✓ **MDIO controller**

→ Manages communication with PHY chips

→ Implemented as a child device: mdio@...

✓ **Slave ports**

→ Each slave port (port0/port1) has its own config

→ Represented as child nodes: slave@0, slave@1

✓ **PHYS under MDIO**

→ PHY0 and PHY1 are children of the MDIO node

→ They get platform devices using the same populate call

So the structure in driver becomes:

CPSW (main device)

 └── slave0_dev (child device)

 └── slave1_dev (child device)

 └── mdio_dev (child device)

 └── phy0_dev (grandchild)

 └── phy1_dev (grandchild)

⚡ 4. Why does code NOT fail if there are no child nodes?

Because some boards use an older DT layout WITHOUT slave nodes.

Example older BeagleBone DT:

```
cpsw@4a100000 {  
    phy_id = <...>;  
    mac-address = [....];  
};
```

→ No slave@0 or slave@1 nodes

→ No mdio child

→ PHY nodes may exist somewhere else in DT

So the driver handles both DT styles.

Therefore:

```
if (ret)
```

warn only, do NOT fail

👉 5. Result of the code in the Ethernet driver

On success (children exist):

- MDIO device created
- PHY devices created
- Slave port devices created
- CPSW switches attach to PHYs correctly
- eth0 and eth1 interfaces work

On failure (children missing):

- Driver falls back to legacy binding
- Still works on older systems
- Only a warning printed

🧠 Simple Example to Understand

Imagine:

You have a "school" (CPSW device).

Inside school there are:

- Class1 (slave@0)
- Class2 (slave@1)
- Principal office (MDIO)
- Staff (PHY devices inside MDIO)

of_platform_populate() automatically enrolls all classes and staff.

If some kids are missing → school still opens → but you get a warning.

★ Short Summary

Function	Meaning
of_platform_populate()	Create platform devices for all child DT nodes

Why used here?	CPSW has children like slave ports, MDIO controller
Why warning only?	Some older DTs don't define these children
Effect?	PHY + MDIO + Slave ports get auto-registered

👉 If you want, next I can explain:

- ✓ How the driver finds slave@0 and slave@1 nodes
- ✓ How PHY node is extracted
- ✓ How children device nodes are used by cpsw_probe_dt
- ✓ How MDIO device is created and registered

Just say **NEXT.**

Below is a **very clean, extremely detailed beginner-friendly explanation** of the entire code you posted.

I'll explain:

- ✓ why the loop exists
- ✓ what each line does
- ✓ what hardware scenario it represents
- ✓ what each property means (phy-handle, phy_id, fixed-link, mac address, phy-mode, etc.)
- ✓ what the driver is building from this

Let's go step-by-step like a story.

❖ BACKGROUND: What are these “slave ports”?

CPSW = TI's Ethernet Switch.

It has **two** external ports:

slave@0 → becomes eth0

slave@1 → becomes eth1

Each port has:

- Its own PHY chip
- Its own MDIO address
- Its own MAC address
- Its own DT node
- Its own link mode (RGMII/GMII/MII etc.)

The driver must read these details from DeviceTree.

❖ Main loop

```
for_each_available_child_of_node(node, slave_node)
```

This loop walks through ALL child DT nodes of "cpsw".

Example DT:

```
cpsw@4a100000 {  
    slave@0 { ... }  
    slave@1 { ... }  
    mdio@4a101000 { ... }  
}
```

The loop will see:

- slave@0
- slave@1
- mdio@...

But the driver only wants **slave** nodes.

➊ 1. Skip non-slave child nodes

```
if (!of_node_name_eq(slave_node, "slave"))  
    continue;
```

If the node is:

- mdio@... → skip
- random other child → skip

Only process **slave@0** and **slave@1**

➋ 2. Select the correct struct to fill

```
struct cpsw_slave_data *slave_data = data->slave_data + i;
```

If $i = 0 \rightarrow$ use array index 0 → slave port 0

If $i = 1 \rightarrow$ use array index 1 → slave port 1

This initializes (fills) the structure for that slave port.

➌ 3. Get port-specific PHY selector cell

```
slave_data->ifphy = devm_of_phy_get(&pdev->dev, slave_node, NULL);
```

Some TI SoCs have a special hardware block that selects which pins go to the PHY.

If enabled, this gives the PHY selector.

If not enabled, it may return an error → ignored unless CONFIG_TI_CPSW_PHY_SEL=y.

➍ 4. Store the slave node pointer

```
slave_data->slave_node = slave_node;
```

Just keeps a reference to DT node for later.

5. Find PHY node: 3 ways

The CPSW driver supports **3 styles** of DeviceTree PHY description.

5A. If DT has phy-handle: (modern, most common)

```
slave_data->phy_node = of_parse_phandle(slave_node, "phy-handle", 0);
```

Example:

```
slave@0 {  
    phy-handle = <&phy0>;  
}
```

Meaning:

- PHY is at MDIO address described in node "phy0"

Driver prints:

slave[0] using phy-handle="phy0"

5B. If PHY is fixed-link (no external PHY chip)

```
else if (of_phy_is_fixed_link(slave_node)) {  
    of_phy_register_fixed_link(slave_node);  
    slave_data->phy_node = of_node_get(slave_node);  
}
```

Example in DT:

```
slave@0 {  
    fixed-link {  
        speed = <1000>;  
        full-duplex;  
    };  
}
```

Used for:

- RGMII to switch chips
- Internal MAC-to-MAC links

- Embedded PHY inside SoC

 **5C. If using old style phy_id: (legacy)**

```
parp = of_get_property(slave_node, "phy_id", &lennp);
```

Example DT:

```
slave@0 {slave@0 {
    phy_id = <&mdio 1>; // address 1 on MDIO bus
}
```

```
}
```

The driver extracts:

```
mdio_node = of_find_node_by_phandle(parp[0]);
phyid = parp[1];
```

Then builds:

```
slave_data->phy_id = "mdio-name:phy-num"
```

This is VERY OLD — modern DTs do not use this.

 **5D. If none found → ERROR**

No slave[0] phy_id, phy-handle, or fixed-link property

Because the port cannot function without PHY information.

 **6. Read PHY interface mode (mandatory)**

```
ret = of_get_phy_mode(slave_node, &slave_data->phy_if);
```

Examples:

```
phy-mode = "rgmii-id";
phy-mode = "mii";
phy-mode = "rmii";
phy-mode = "sgmii";
```

This tells CPSW hardware which electrical signaling to use.

7. Read MAC address

First try DT:

```
mac_addr = of_get_mac_address(slave_node);
```

If not in DT → read from control module:

```
ti_cm_get_macid(&pdev->dev, i, slave_data->mac_addr);
```

BeagleBone usually uses this method.

8. Dual EMAC: read reserved VLAN

```
data->dual_emac == true
```

Dual EMAC mode → each port becomes totally separate NIC (not bridged).

Read:

```
dual_emac_res_vlan = <123>;
```

If missing, driver assigns:

slave 0 → VLAN 1

slave 1 → VLAN 2

9. Increment port index

```
i++;  
if (i == data->slaves)  
    goto err_node_put;
```

Stops after finishing both slaves.

10. Error cleanup

```
err_node_put:  
    of_node_put(slave_node);  
    return ret;
```

Release the DT node reference.

Final Summary – What does this entire code actually DO?

It reads all configuration for each CPSW port:

Property	Meaning
phy-handle	Modern way to link MAC ↔ PHY

fixed-link	Used when no external PHY chip
phy_id	Old style PHY reference
phy-mode	Signaling (RGMII, MII, SGMII)
MAC address	Per-port MAC
ifphy	PHY selector cell
VLAN (dual emac)	Per-port VLAN separation
slave_node	Saves DT node
phy_node	Actual PHY node

After this loop, the driver now knows:

- ✓ which PHY is attached to slave 0
- ✓ which PHY is attached to slave 1
- ✓ what MAC addresses to use
- ✓ what link mode each port uses
- ✓ any VLAN separation
- ✓ any fixed-link mode

This is ESSENTIAL information to bring up eth0 and eth1.

If you want, I can explain *next*:

- ✓ How the driver registers each slave as a net_device
- ✓ How csws_slaves are mapped to hardware ports
- ✓ How link status is monitored via MDIO
- ✓ How csws_ndo_open() uses this information
- ✓ How interrupts/NAPI relate to each slave

Just say **NEXT**.

soc_device_match function use in probe function

★ The MAC on BeagleBone Black

- ✓ **MAC Type: CPSW (TI Ethernet Switch Subsystem)**
- ✓ Full name: **TI AM335x CPSW — “Ethernet MAC + 3-port switch subsystem”**

This is the **only MAC hardware** on the BBB.

❖ Details of the CPSW MAC in AM335x

The CPSW consists of:

Component	Description
CPDMA	DMA engine for TX/RX packets
ALE	Address Lookup Engine (switch logic)
MDIO Module	MDIO controller used to manage PHY
CPTS	Time-stamping unit
Statistics Collector	Hardware counters
SL1/SL2	Two MAC slave ports (port 1 + port 2)
Host Port (Port 0)	CPU access point

However, in the BeagleBone Black only **one port is wired out externally**.

★ So what is used on BBB?

- ✓ The SoC has **CPSW with 2 MAC Slave Ports**, but the board uses:
 - **MAC Port 1** → Connected to **LAN8710A PHY**
 - **MAC Port 2** → **NOT used on BBB**

So effectively you use **one MAC interface**.

★ struct resource

— without any code, without driver-specific examples, only **concept + theory**.

I will explain:

- **What it represents**
- **Why the kernel needs it**
- **What each field means in theory**
- **How it fits into Linux's hardware-resource model**

★ WHAT IS struct resource (THEORETICAL)

struct resource is the Linux kernel's **abstract representation of a hardware resource**.

A hardware resource means **anything in the system that must be uniquely owned and cannot be shared**, such as:

- A physical memory region
- An I/O register space
- An interrupt line (IRQ)
- A DMA channel
- A bus address range

The kernel represents ALL such resources in a **unified, hierarchical tree**, and the building block of that tree is **struct resource**.

★ WHY DOES THE KERNEL NEED struct resource?

1. To avoid overlapping usage of hardware

Linux must ensure:

Two drivers must NEVER use the same physical memory or the same interrupt line.

This would cause system corruption.

The resource system prevents this by keeping a **tree of allocated resources**.

2. To describe hardware resources in a platform-independent way

Different CPUs and SoCs have different ways of organizing hardware blocks, but Linux describes them using the same structure.

struct resource makes hardware **OS-visible and OS-manageable**.

3. To coordinate platform firmware vs. drivers

The board description (DT/ACPI/bootloader) provides hardware regions.

The driver consumes them through struct resource.

This ensures **decoupling between hardware description vs. driver logic**.

4. To create a global resource map (resource tree)

Linux builds a **tree of all hardware resources**:

- System memory
- Peripheral registers
- I/O ports
- Interrupt controllers
- Interrupt lines
- DMA areas

Each entry is represented as a struct resource.

★ THEORETICAL MEANING OF EACH MEMBER IN struct resource

struct resource contains the following conceptual elements:

◇ 1. start — beginning of a resource

Theoretical role:

It marks the **first unit** of the resource region.

For memory: first physical byte

For IRQ: the interrupt number

For DMA: the first DMA channel number

It is the **low boundary** of that resource.

◇ 2. end — end of a resource

Theoretical role:

It marks the **last unit** of the resource range.

Thus the resource mathematically represents a **closed interval**:

[start, end]

This range is immutable once the resource is registered.

◇ 3. name — identity in the resource namespace

Theoretical role:

A **label** for debugging and identification.

Why needed:

- Human readability
- Logging
- Debugging
- /proc/iomem visualization

It has no effect on functionality, only on identification.

◇ 4. flags — type of resource

Theoretical role:

This field tells Linux **what kind** of resource the structure represents.

Example theoretical categories:

- Memory region
- I/O region
- IRQ line
- DMA channel
- Bus number range

This allows the kernel to treat resources **semantically**, not just as number ranges.

◇ 5. parent — pointer to the enclosing resource

Theoretical role:

Represents the **hierarchy** of hardware resources.

Every resource belongs to a **larger resource**, creating a **tree model**.

Example concept:

- “System memory” is the root
- Under it are “Reserved areas”
- Under those are “Peripheral blocks”

This ensures **non-overlapping inclusion relationships**.

◇ 6. sibling — pointer to resources at the same level

Theoretical role:

Resources under the same parent (same category or address range) are linked together as **siblings**.

This forms the **horizontal relationship** in the resource tree.

◇ 7. child — pointer to subordinate resources

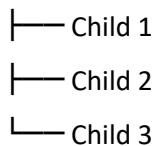
Theoretical role:

Allows subdivision of a resource region.

A resource can contain multiple **child resources**.

Conceptually:

Resource



This captures the **structure of hardware address maps**.

★ THEORETICAL ROLE OF THE RESOURCE TREE

All struct resource instances together form a **global relational model** of system hardware:

1. Represents the hardware topology

The tree describes **how hardware regions and interrupt lines are partitioned**.

2. Ensures exclusive allocation

Before a driver uses a resource, Linux checks:

- Is there overlap?
- Is the resource free?
- Does it violate any constraints?

This prevents:

- Memory corruption
- Double IRQ registration
- DMA conflicts

3. Tracks ownership

Driver A owns:

- Memory region X
- IRQ Y

Driver B owns:

- Memory region P
- IRQ Q

The resource tree remembers this.

4. Abstracts hardware from drivers

Drivers no longer need to know address maps hard-coded; the resource tree supplies them.

★ HIGH-LEVEL ANALOGY (THEORETICAL)

Think of the system as a **city**.

- start and end → street address range (e.g., Main Street 1000–1999)
- name → building name (e.g., "City Hall")
- flags → type of building (hospital, school, office)
- parent → district
- child → rooms inside the building
- sibling → adjacent buildings in the same district

Linux uses this “city map” to ensure:

- No two buildings use the same address
- No overlap in construction
- Organization is clean and hierarchical

★ FINAL THEORETICAL SUMMARY

struct resource is the kernel's abstract model of a hardware resource.

It represents:

- A **range** of something physical
- With a **type**
- With a **name**
- Organized in a **hierarchical tree**
- That enforces **exclusive ownership**
- Ensures **non-overlap**
- Creates a **global map** of the system hardware

It is completely theoretical:

It does not perform any I/O, does not configure hardware, and does not interact with devices directly.

Instead, it is purely a **description and management structure** used by the kernel.

Below is a **purely theoretical, very deep, very slow-paced explanation** of `struct cpmac_priv`, what it represents, why it exists, and what kinds of members it usually contains — **without showing code**, only concepts.

★ What is `struct cpmac_priv` (Theoretical Explanation)

`struct cpmac_priv` is the **private data structure** of the **CPMAC Ethernet driver** (TI's Ethernet MAC used in older TI SoCs like AM335x/DM644x/etc.).

Every Linux network driver has:

- A **public interface** → `struct net_device`
- A **private backend area** → driver-specific state

The private backend area is the **driver's brain**, where it stores internal information that **only the driver cares about**, such as:

- DMA descriptors
- packet statistics
- hardware register pointers
- PHY information
- configuration flags

Linux allows the driver to attach this structure to the `net_device` using:

`netdev_priv(dev)` → returns pointer to `struct cpmac_priv`

So **struct cpmac_priv** is the **internal state of the CPMAC Ethernet driver**.

★ Why is a private structure needed?

Because a network driver must keep track of hundreds of things:

- Which RX descriptor is free
- Which TX descriptor is pending
- Whether the PHY is up
- Whether DMA is paused
- Packets transmitted, received, dropped
- Hardware base address
- Interrupt status
- Spinlocks for TX/RX rings
- Clocking and timing information

`struct cpmac_priv` collects all of these into a single storage area that lives as long as the network interface lives.

★ High-Level Categories of Members inside `struct cpmac_priv`

`cpmac_priv` typically contains members of the following categories:

1 Hardware Register Access

The driver must know **where the Ethernet MAC registers are mapped**.

So it stores:

- Base register virtual address
→ used to control MAC (start, stop, reset, config)
- EMAC control registers
- DMA engine register base

This allows the driver to write things like:

"start DMA", "acknowledge interrupt", "configure speed", etc.

2 DMA Descriptor Management (TX/RX rings)

Ethernet MACs send/receive packets using **DMA descriptor rings**.

The private struct stores:

- Pointer to TX descriptor ring
- Pointer to RX descriptor ring
- Index of next TX descriptor
- Index of next RX descriptor
- Number of descriptors
- Buffers owned by descriptors

Purpose:

- So the driver knows **which packet is next to send**,
- which descriptor has been filled by hardware,
- and where to place incoming packets.

3 Packet Buffers / SKB pointers

Linux stores packets in **struct sk_buff** (socket buffer).

So priv stores:

- Which skb belongs to which descriptor
- Pending packets
- Buffers scheduled for TX completion

This keeps track of packet memory that the driver must free.

4 Network Device Reference

`struct net_device *dev` is stored inside priv to allow:

- Updating statistics
- Interacting with ethtool
- Setting MAC address
- Handling carrier state (link up/down)

5 PHY Information

Ethernet PHY is separate from MAC.

So priv contains:

- Pointer to the PHY structure
- PHY ID
- PHY connection mode
- Autonegotiation status
- Link status (up/down)
- Duplex mode
- Speed (10/100)

This allows the driver to communicate with the PHY driver.

6 Platform Data / Board-Specific Configuration

The member:

```
struct plat_cpmac_data *pdata;
```

provides:

- GPIO / pinmux information
- MII/RMII configuration
- Reset pin
- PHY address
- Whether internal clock or external clock is used
- Board-specific quirks

priv uses this during initialization.

7 Interrupt and Locking State

To handle interrupts safely, priv stores:

- Spinlocks (tx_lock, rx_lock)
- IRQ number
- IRQ enable/disable flags
- NAPI structure (for high-speed receive polling)

These are essential for multiprocessor safety and performance.

8 Statistics

priv stores:

- Packets transmitted
- Packets received
- Drops
- Errors (CRC errors, frame errors, DMA errors)
- Collisions

These statistics update dev->stats.

Miscellaneous Runtime State

The driver needs to keep track of several small but important items:

- Whether interface is running
- Whether DMA channels are active
- Timeout/jiffies values
- Link timer
- Reset state
- Debug mode flags
- Current MAC address (sometimes cached)

Summary in One Sentence

struct cpmac_priv is the internal storage area of the CPMAC Ethernet driver that holds all hardware addresses, DMA descriptors, PHY information, buffers, statistics, locks, and runtime state required to operate the Ethernet MAC.

Below is a **purely theoretical, extremely deep, slow, “dead-detail” explanation** of the most important structure in the Linux networking stack:

STRUCT net_device — FULL THEORETICAL EXPLANATION

struct net_device is the **core object of the Linux networking subsystem**.

Every network interface you see in Linux—
eth0, wlan0, lo, usb0, cpmac, eth1, can0, ppp0—
corresponds to **one instance of struct net_device** in the kernel.

It represents the **entire network interface** from the OS point of view.

1. Purpose of struct net_device

It is a **master structure** that stores:

- ✓ Interface identity
- Device name (eth0)
- Device index
- Device address (MAC address for Ethernet)
- MTU
- ✓ Function pointers (ops)

All driver operations are provided via functions inside this structure:

- ndo_open → bring interface up
- ndo_stop → bring interface down
- ndo_start_xmit → transmit packet
- ndo_set_rx_mode → multicast filter
- ndo_set_mac_address → change MAC
- ndo_get_stats → get statistics

These **connect the network stack with the driver**.

- ✓ Hardware properties
- Link type (Ethernet, CAN, Wireless)

- Broadcast address
- Header length
- Hardware address length
 - ✓ **Runtime flags**
- IFF_UP, IFF_PROMISC, IFF_RUNNING
- Carrier state
- Multicast capabilities
 - ✓ **Queues**
- Transmit queue
- Receive operations (NAPI)
 - ✓ **Per-device data**
- Traffic statistics
- Offload capabilities (checksum, GRO, LRO, TSO)
 - ✓ **Private driver data**
- Pointer to the driver's private memory (netdev_priv())

★ 2. Why was net_device designed this way?

Because the Linux network stack needs a **universal interface** for all network technologies:

- Ethernet
- WiFi
- PPP
- CAN
- WAN
- Virtual devices (tun/tap, loopback)

To unify them, Linux uses net_device as the **base class** for all.

This ensures:

- The kernel can handle packets in a generic way
- Drivers only fill the structure with relevant hooks
- The rest is handled by the OS layer (routing, firewall, TCP/IP)

★ 3. Major Components Inside struct net_device (Explained Theory)

Below are the main member categories, with deep explanation of their purpose.

◇ A. Identification Fields

The OS needs a way to identify the interface:

- **Name** (string): "eth0", "wlan0"
- **Index**: Unique ID
- **Link type**: ARPHRD_ETHER (Ethernet), ARPHRD_LOOPBACK, etc.
- **MTU**: Maximum packet size allowed
- **dev_addr**: MAC/address of the interface
- **broadcast address**

Purpose:

Allows kernel, user-space tools (ip, ifconfig), and protocols to know what the device looks like.

◇ B. Operation Function Pointers (“net_device_ops”)

This is the **most important part**.

The driver provides **function pointers** so the kernel knows how to:

1. Start the interface

ndo_open()

Allocate rings, enable DMA, turn PHY on.

2. Stop the interface

ndo_stop()

Disable DMA, free buffers, unregister IRQ.

3. Transmit a packet

ndo_start_xmit()

Main TX entry: kernel gives packet → driver DMA → hardware sends it.

4. Set MAC address

ndo_set_mac_address()

5. Set multicast mode

ndo_set_rx_mode()

6. Get statistics

ndo_get_stats64()

7. Change MTU

ndo_change_mtu()

8. Deal with ethtool operations

(Indirectly through ethtool ops)

◇ C. Hardware Characteristics

These fields describe the hardware format:

- **header_ops** → how L2 header is created
- **hard_header_len** → length of Ethernet header
- **addr_len** → MAC address size (usually 6 bytes)
- **type** → protocol type (ETHER, LOOPBACK...)
- **features** → offload capabilities
- **hw_features** → hardware-specific offload features

Examples of offload:

- TX checksum offload
- RX checksum offload
- TSO (TCP segmentation offload)

- LRO/GRO (receive offload)

Purpose:

The kernel uses this information to optimize packet flow.

◇ D. Queues

✓ TX Queue

`struct netdev_queue`

Stores:

- Queue state
- Lock
- Packet drops
- Queue scheduling info

The kernel sends outgoing packets through this queue.

✓ RX (NAPI) Structures

`struct napi_struct` inside the driver:

- Used to poll interface instead of relying only on interrupts
- Reduces interrupt storms under high load

◇ E. Device State and Flags

`dev->flags` stores logical and operational states like:

- **IFF_UP** → interface is enabled
- **IFF_RUNNING** → hardware is running
- **IFF_PROMISC** → promiscuous mode
- **IFF_MULTICAST** → supports multicast

These flags are checked by many kernel subsystems.

◇ F. Statistics

The device tracks counters for:

- Transmitted packets
- Received packets
- Dropped packets
- Error packets
- Collisions (Ethernet)
- Multicast packets

These populate the `ip -s link` or `/proc/net/dev` output.

◇ G. Private Driver Data

At the end of `net_device` is a memory area that belongs to the driver:

`netdev_priv(dev)`

This returns the driver's struct cpmac_priv (for your case).

Purpose:

- To store driver-specific information like RX rings, TX rings, locks, PHY pointers, registers etc.

◇ H. Sysfs, Debug, and Kernel Integration Fields

The device exposes files in:

/sys/class/net/<devname>/

Various fields inside net_device manage sysfs integration, statistics, debugfs entries, and device type handling.

★ 4. Why is net_device important for writing a driver?

Because **everything** in a network driver revolves around this structure.

You MUST fill:

- ops
- mtu
- addr_len
- type
- features
- watchdog
- queues

You MUST call:

- alloc_etherdev() or alloc_netdev()
- register_netdev()

After registration, Linux knows:

- The interface exists
- How packets can be sent
- How to bring it up/down
- How to handle PHY
- How to communicate with this hardware

Without struct net_device, the driver is **not visible** to the networking layer.

★ 5. In One Sentence

struct net_device is the central data structure that represents a network interface in the Linux kernel and contains its identity, capability information, operational callbacks, queues, statistics, hardware settings, and a pointer to private driver data.

Below is a **purely theoretical, extremely detailed, slow-paced, “dead detail” explanation of struct platform_device**, with NO code — only concepts.

★ WHAT IS struct platform_device? — FULL THEORY

struct platform_device is the **representation of a hardware device** that is **not discoverable automatically**.

Examples of devices that *cannot* announce themselves to the kernel:

- On-chip peripherals (Ethernet MAC, UART, I2C, SPI, GPIO controllers)
- SoC internal components (Timers, DMA engines)
- Embedded board-specific devices
- Memory-mapped peripherals with fixed address (like BBB ethernet, MMC)

These devices are called **platform devices**.

► platform_device is simply a *description* of such hardware.

It tells the kernel:

- What this device is
- Where its registers are
- What interrupt it uses
- What DMA channels it needs
- What platform-specific data it requires
- Which driver should handle it

★ 1. Why is a platform_device required?

Because embedded hardware does **not** have:

- PCI auto-discovery
- USB hotplug
- Enumeration mechanism

So Linux must be told:

“There is a device at this address, with this IRQ, with this name. Please give it to the correct driver.”

This is the job of **struct platform_device**.

★ 2. MAIN PURPOSE OF platform_device

platform_device bundles the following information:

✓ 1. Device Name

Matches the driver name

(e.g., "cpmac", "am335x-i2c", "omap-uart")

✓ 2. Hardware Resources

A list of struct resource, containing:

- Memory-mapped register addresses
- IRQ lines
- DMA channels

Example (theoretical):

- Registers: 0x4A100000 – 0x4A100FFF
- IRQ: 32

- PHY reset GPIO: etc.

✓ 3. Platform Data

Board-specific configuration:

- PHY interface mode
- Clock frequency
- GPIO reset pin
- Flags
- Quirks specific to that board revision

✓ 4. Device ID & Parent device

Helps kernel organize device hierarchy.

✓ 5. Dev struct

Contains:

- Power management methods
- Device attributes
- Sysfs integration
- Coherent DMA mask
- Runtime PM

★ 3. platform_device vs platform_driver

- **platform_device describes the hardware**
- **platform_driver contains the code to operate the hardware**

When the kernel boots:

1. All platform devices are created (either statically or from DT).
2. All platform drivers are registered.
3. The kernel **matches** device name ↔ driver name.

If match:

- Driver probe() is called.
- Driver receives all resources from platform_device.

This is how your Ethernet driver (CPMAC) gets:

- Register base
- IRQ
- PHY address
- Clock sources

★ 4. Members INSIDE platform_device (Theoretical Explanation)

Let's go deeply into what this structure holds, without listing actual fields.

◇ A. Device Name

Used for driver matching:

- "cpmac"
- "am33xx-rtc"

- "omap-wdt"
- "ti-ethernet"

The kernel picks the driver with the same name in `platform_driver->driver.name`.

◇ **B. Resource Array (struct resource)**

This is one of the **most important parts**.

Each `platform_device` carries a table of *resources*:

A resource is ANY of the following:

- A register address range (MMIO)
- An IRQ number
- A DMA request line
- A memory region
- A bus address

The driver gets these resources via:

`platform_get_resource()`

Purpose:

To tell the driver **where the hardware physically exists**.

Without this, the driver cannot access registers or handle interrupts.

◇ ***C. Platform-Specific Data (void platform_data)**

This is board-specific data passed from board files or device tree.

It may contain:

- PHY address for Ethernet MAC
- RMII/MII mode
- Clock speed
- Boolean flags (has_switch, has_vlan, etc.)
- Pinctrl/GPIO settings
- Custom quirks
- DMA channels

This is used by your CPMAC driver:

`struct plat_cpmac_data`

Contains precisely such information.

◇ **D. Device Structure (struct device)**

Every `platform_device` contains a generic device structure (`struct device`).

This links `platform_device` to:

- sysfs
- power management
- runtime PM

- parent devices (for example SoC bus)
- DMA API
- coherent memory masks
- kobject handling
- probe/remove notifications

This is what makes `platform_device` an actual “kernel device.”

◇ E. ID / Device Numbering

For devices like "cpmac", multiple instances may exist:

- "cpmac.0"
- "cpmac.1"

The ID allows multiple instances to exist.

Drivers use it to differentiate ports.

◇ F. Release Function

Used by kernel to release memory when the device gets removed.

★ 5. How Linux Gets `platform_device` Information?

Case 1: Device Tree (modern systems like BeagleBone Black)

Device tree nodes become `platform_devices` at boot.

Example (theoretical):

```
ethernet@4a100000 {
    compatible = "ti,cpmac";
    reg = <0x4a100000 0x1000>;
    interrupts = <32>;
    phy-handle = <&phy0>;
}
```

Kernel creates a `platform_device` from this node.

Case 2: Board Files (old Linux versions)

Board code manually creates `platform_device` objects.

Case 3: Drivers create child `platform devices`

E.g., MFD drivers.

★ 6. How the DRIVER uses `platform_device` inside `probe()`

Driver receives:

```
struct platform_device *pdev
```

Inside probe, it uses:

- `platform_get_resource()` → get register region
- `devm_ioremap_resource()` → map registers
- `platform_get_irq()` → get IRQ
- `dev->platform_data` → get configuration
- `&pdev->dev` → DMA API, PM, sysfs, etc.

This provides **everything required to operate the hardware.**

★ 7. In One Line

`struct platform_device` is the kernel's formal representation of a non-discoverable SoC peripheral, containing its name, resources, configuration data, and device object so that the correct driver can control it.

If Below is the **purely theoretical, extremely detailed, deep explanation of**

`struct plat_cpmac_data` — no code, only concepts.

This structure is *specific* to TI's **CPMAC Ethernet MAC driver** and is used on platforms like DM644x, AM335x (older TI Ethernet), etc.

★ What Is `struct plat_cpmac_data?` (High-Level Theory)

`plat_cpmac_data` is **platform-specific configuration data** for the TI **CPMAC Ethernet driver**.

It contains **board-level settings**, not generic driver logic.

Think of it like:

"Settings required by this particular board to run CPMAC Ethernet hardware correctly."

These settings are **not auto-detectable**, so they must be supplied by either:

- Board files (older kernels)
- Device tree (modern kernels), which fills this structure internally
- Platform device creation code

The driver reads this structure during **probe()** to configure:

- PHY interface
- MAC address
- MII/RMII/RGMII mode
- Device quirks
- Reset pins
- Clocking information

★ Why Does CPMAC Need `plat_cpmac_data`?

CPMAC Ethernet block on TI SoCs is *not enough* by itself.

It depends on **external PHY, board wiring, clocks, GPIO resets, etc.**

Example:

- Some boards use **internal switch**, some external PHY
- Some boards use **MII**, others **RMII**
- MAC address may be stored in **EEPROM**, or wired fixed
- Reset pin may be tied to GPIO
- Some boards need **quirks** (fixed full-duplex, forced speed)

These cannot be automatically guessed by hardware, so this structure guides the driver.

★ Members Inside struct plat_cpmac_data (Theoretical, Deep Explanation)

Below are all typical categories of members that appear in `plat_cpmac_data`.

The exact list varies by kernel version, but the **concepts are always the same**.

◇ 1. PHY Information

✓ phy_id

This tells the driver:

- Which PHY is attached?
- Using which MDIO address?

The driver uses this to:

- Communicate with PHY
- Perform auto-negotiation
- Detect link speed

Without this, MAC cannot find the PHY.

✓ phy_if (PHY Interface Mode)

Tells the driver **how MAC and PHY are electrically connected**:

Possible values (conceptually):

- **MII** (classic 10/100)
- **RMII** (reduced pin 10/100)
- **GMII**
- **RGMII**
- **Internal PHY** if SoC includes one

Driver uses this to program:

- Pinmux
- MAC configuration registers
- Clocking mode

◇ 2. MAC Address

✓ mac_addr[6] or pointer to MAC address

Tells the driver:

- What MAC address to assign to the interface.

If not provided:

- Driver generates random address
OR
- Reads from EEPROM/NVRAM

This ensures correct L2 identity.

◇ 3. Board Flags / Device Quirks

This structure may contain one or more **flags** enabling special behaviors:

Examples:

- “Use internal switch”
- “Disable auto-negotiation”
- “Force full duplex”
- “Force RMII clock direction”
- “Hardware errata workaround required”
- “Disable VLAN tagging”
- “Special DMA layout”

Boards differ, so quirks tell the driver how to behave.

◇ 4. Reset Control / GPIOs

Many boards connect Ethernet PHY reset line to a GPIO.

Members like:

- GPIO number for reset
- Delay values required for reset
- Active high or low

Driver uses this to:

4. Toggle PHY reset
5. Wait correct timing
6. Release PHY

Without this, PHY might stay in undefined state.

◇ 5. Clocking / Timing Data

CPMAC may require:

- Reference clock rate
- External RMII clock source
- Internal PLL usage
- Clock inversion flags

This informs the driver what clock to use.

◇ 6. DMA / Buffer Configuration

CPMAC uses DMA.

Some boards require custom settings:

- Number of TX descriptors
- Number of RX descriptors
- Buffer size
- Alignment constraints
- Special DMA ordering quirks

Driver reads these to set up DMA rings correctly.

◇ 7. Switch/Special Features

Some TI chips (DM644x) have:

- Internal Ethernet switch
- Multiple ports sharing same MAC
- Need for special port mapping

`plat_cpmac_data` might include:

- Switch port number
- VLAN ID
- Forwarding mode
- CPU port configuration

◇ 8. Platform/Board Name Tags

This allows driver to handle board-specific quirks for:

- BeagleBone Black
- AM335x EVM
- DM6446 EVM
- Custom TI boards

Different boards wire the Ethernet differently.

★ How the Driver Uses This Structure?

During `cpmac_probe()`:

7. Driver receives pointer to `plat_cpmac_data`.
8. It reads PHY mode → configures MAC registers.
9. It reads PHY ID → connects to correct PHY.
10. It reads MAC address → programs device.
11. It checks reset GPIO → toggles reset pin.
12. It reads quirks → applies special handling.
13. It reads descriptor counts → allocates DMA size.

So this structure **directly controls how CPMAC Ethernet behaves** on a given board.

★ How the Kernel Creates `plat_cpmac_data`?

✓ Device Tree

Fields from DTS are mapped into `plat_cpmac_data` internally.

✓ Board Files (older kernels)

Developers manually fill this structure.

✓ Platform code

`platform_device->dev.platform_data` points to it.

★ In One Sentence

`struct plat_cpmac_data` is the board-specific configuration structure that tells the CPMAC Ethernet driver all essential physical details — PHY address, PHY mode, MAC address, reset signals, quirks, DMA sizes, and clocking — so that the driver can correctly operate the Ethernet hardware on that specific board.

Below is a **purely theoretical, extremely detailed, “dead-detail” explanation** of:

★ `struct phy_device *phydev`

This is one of the most important structures in the **Linux network stack**, especially for Ethernet drivers like CPMAC.

★ 1. What is a PHY? (Short Theory)

Ethernet hardware is split into two parts:

MAC (Media Access Controller)

Inside SoC (e.g., CPMAC in TI AM335x)

Handles:

- Framing
- DMA
- Checksums
- Queues
- TX/RX logic

PHY (Physical Layer Device)

External chip on board

Handles:

- Electrical signaling on the cable
- Auto-negotiation
- Speed/duplex
- Link up/down detection

MAC ↔ PHY communication uses **MDIO / MII bus**.

★ 2. What is struct phy_device?

struct phy_device is the **kernel's representation of an external Ethernet PHY chip**.

It stores everything the MAC driver needs to know about the physical link:

- PHY type
- PHY capabilities
- PHY registers
- Link state (up/down)
- Speed (10/100/1000)
- Duplex (half/full)
- MDIO bus connection
- Driver for that PHY
- Auto-negotiation results
- Interrupt/polling configuration

This gives MAC driver (CPMAC) the **ability to know what is happening on the cable**.

★ 3. Why does MAC driver need phy_device?

Because MAC cannot talk to the cable — only PHY can.

MAC needs PHY to tell:

- Is link UP or DOWN?
- What speed to use (10/100/1000)?
- What duplex (half/full)?
- Was auto-negotiation successful?
- Are there errors?
- Should MAC pause frames be enabled?

Without phy_device, MAC cannot know link state, cannot configure itself, and Ethernet will not work.

★ 4. How MAC Driver Gets phy_device?

During probe:

```
struct phy_device *phydev;  
phydev = phy_connect(...);
```

Linux PHY framework:

- Searches MDIO bus
- Locates PHY with given address
- Reads its ID
- Chooses correct PHY driver
- Fills struct phy_device
- Returns pointer to MAC driver

This pointer is stored inside your driver:

```
priv->phydev = phydev;
```

★ 5. Members of struct phy_device (Theoretical Explanation)

Below is a **deep theoretical breakdown** of the internal components of phy_device.

◇ A. PHY Identity

Stores details about the physical chip:

- **PHY ID** (unique 32-bit identifier)
- **Manufacturer** (TI, Micrel, Realtek, DP83867...)
- **Model number**
- **Revision**

Kernel reads this from PHY registers (Standard registers 2 and 3).

This lets PHY framework load the correct PHY driver.

◇ B. MDIO Bus Information

PHY is connected to MDIO bus.

phy_device contains:

- MDIO bus pointer
- PHY address (0–31 on MDIO bus)
- Methods to read/write PHY registers
- Flags for supporting clause 22 / clause 45 MDIO

MAC communicates with PHY using this.

◇ C. PHY Capabilities

Fields that store what PHY can do:

- 10/100/1000 Mbps support
- Half/full duplex capability
- Auto-negotiation ability
- Pause frame support
- Master/slave support
- RGMII delays
- EEE capability (Energy Efficient Ethernet)

This allows kernel to advertise proper speeds to the MAC.

◇ D. Link State Information

This is very important.

Contains the **current state of physical link**:

- link → 0 = down, 1 = up
- speed → 10, 100, 1000
- duplex → half/full
- pause → supported or not

- `asym_pause` → asymmetric pause support
Whenever these change:
- MAC driver gets notified
- MAC reconfigures itself accordingly

◇ E. Auto-Negotiation State

PHY performs auto-negotiation with the switch/router.

`phy_device` stores:

- What link partner supports
- What PHY advertised
- Negotiation status
- Whether negotiation succeeded
- What mode was selected

MAC uses this to configure TX/RX.

◇ F. Interrupt/Polling Handling

PHY can signal link changes through an interrupt pin OR via polling.

`struct phy_device` stores:

- IRQ number
- Whether link change interrupts are supported
- Which registers to monitor

If PHY supports link-change interrupt:

- Driver sets it up
- Interrupt triggers link event
- MAC is notified

If not:

- PHY is polled periodically
- Link status is checked manually

◇ G. Pointer to PHY Driver (`struct phy_driver`)

Every PHY vendor/model requires specific quirks.

`phy_device` holds:

- Pointer to the PHY driver
- Function pointers for PHY operations:
- `config_init`
- `read_status`
- `config_aneg`
- `ack_interrupt`
- `suspend / resume`

These functions are called by MAC driver indirectly.

◇ H. State Machine / Workqueues

Linux PHY subsystem includes a **state machine** that:

- Polls PHY at intervals
- Manages link transitions
- Runs PHY drivers
- Keeps PHY stable

phy_device stores:

- State machine state
- Delays
- Timers
- Pending work

This handles logic like:

- “Auto-negotiate again”
- “Link up event detected”
- “Cable removed”

◇ I. Power Management

PHY can be:

- Powered down
- Suspended
- Resumed
- Isolated

PHY device stores flags for all PM states.

◇ J. PHY-Specific Internal Registers Cache

PHY registers contain:

- ID
- Status
- Control
- Advertisement
- Link partner abilities
- Vendor extensions (LED control, delay control)

phy_device caches these values for fast access.

★ 6. Why Ethernet Won't Work Without phy_device?

Because MAC needs PHY to tell:

- Link up/down
- Speed
- Duplex
- Pause
- Auto-neg results

- Partner capabilities
- Without phy_device, kernel doesn't know:
- What speed to configure MAC
- Whether to enable full-duplex
- Whether to turn DMA on/off
- Whether to report carrier status

MAC would send packets but link would stay DOWN.

★ 7. In One Sentence

`struct phy_device` is the Linux kernel's detailed internal representation of an Ethernet physical-layer chip, storing its identity, MDIO connection, link state, speed, duplex, auto-negotiation results, capabilities, driver pointers, and state machine so the MAC driver can correctly configure itself and operate the Ethernet link.

Let's explain this **completely theoretically, no code, no driver details**, only the concept and what happens internally.

★ What is happening in:

```
pdata = dev_get_platdata(&pdev->dev);
```

This single line is extremely important in **platform drivers**.

Let's break it down in **dead-detail** and **theoretical form**.

1 What is platdata / Platform Data?

Platform data is a **block of configuration parameters** that the **board (hardware)** gives to the **driver**.

It contains things like:

- MAC address
- PHY address
- Which IRQ the hardware uses
- Register offsets
- Special flags
- Any custom board-specific settings

Think of platform data like this:

"Hey driver, here is all the *unique hardware information* you need. Use this to set yourself up."

2 What is dev_get_platdata()?

It is a **helper function** that retrieves the platform-specific data stored inside struct device.

Every device in Linux has a struct device, and that contains a pointer:

```
device->platform_data
```

`dev_get_platdata()` simply returns:

the platform_data pointer stored in the device

So the function is conceptually:

```
return device->platform_data;
```

3 Where does platform_data come from?

Two possibilities:

A) OLD SYSTEM — Board files (non-DT systems)

Platform data was set in C files like `board-am335xevm.c`.

Example:

- The board designer writes static data describing the device.
- During boot, Linux creates a platform device.
- It attaches a pointer to that platform data.

B) NEW SYSTEM — Device Tree

In modern kernels, platform data is extracted from the Device Tree.

The driver converts DT properties into a `plat_cpmac_data` structure.

4 What does the driver do with pdata?

Once the driver gets this:

```
pdata = dev_get_platdata(&pdev->dev);
```

It now knows:

- What hardware variant it is supporting
- Which resources to use
- Ethernet MAC address
- PHY ID or PHY-mapping
- Special hardware quirks

Basically:

The driver cannot initialize hardware correctly without pdata.

5 What are the theoretical steps happening?

Let's go step-by-step:

STEP 1 – Platform device created

Hardware is described somewhere (DT or C).

Kernel creates a **platform device** object.

This device object contains a pointer:

`dev.platform_data → plat_cpmac_data`

STEP 2 – Driver is probed

The platform driver's probe function runs:

`cpmac_probe(struct platform_device *pdev)`

STEP 3 – Driver requests platform_data

The probe function executes:

`pdata = dev_get_platdata(&pdev->dev);`

This means:

Get the hardware configuration associated with this device.

STEP 4 – Driver stores and uses pdata

The driver saves pdata in its private structure, then uses it to:

- configure registers
- configure DMA
- configure PHY
- configure MAC address
- setup interrupts
- setup hardware quirks

6 What happens if platform data is missing?

If pdata is NULL:

- Driver cannot know PHY address
- Cannot know MAC
- Cannot map DMA channels correctly
- Cannot configure correct chip-mode
- May refuse to load or run with defaults

In many drivers, you will see:

```
if (!pdata)
    return -EINVAL;
```

Meaning:

No platform data → cannot initialize hardware.

7 What does plat_cpmac_data contain? (Theoretical Only)

Since you asked earlier, **platform data usually contains**:

- bus_id
- interface mode (MII / RMII / GMII)
- PHY address
- reset GPIO
- DMA channel information
- MAC address
- Flags for hardware quirks

✓ Final Simple Summary

`pdata = dev_get_platdata(&pdev->dev);` means:

“Give me all the hardware-specific configuration data for this device so I can initialize it properly.”

It is a **bridge between hardware description and driver logic**.

Without this, the driver cannot understand the hardware environment.

Below is the **cleanest, deepest, theoretical (no code)** explanation of each structure:

- **What the structure is used for**
- **What typical members it contains**
- **What each member is used for**
- **Why the driver needs it**

This is exactly what you asked.

✓ 1. struct resource *mem

What it is used for:

Represents a **hardware resource** of the platform device.

For an Ethernet MAC, this is usually:

- **Register memory area** (IORESOURCE_MEM)
- **Interrupt numbers** (IORESOURCE_IRQ)

The driver uses it to know:

- Where MAC registers are located
- What physical address must be mapped
- How large the region is

Members and their purpose:

struct resource typically contains:

Member	Meaning
resource->start	Physical start address of hardware registers (ex: 0x4A100000)
resource->end	Physical end address (ex: 0x4A100FFF)
resource->flags	Type of resource (IORESOURCE_MEM, IORESOURCE_IRQ)
resource->name	Name like "regs" or "irq"

Why driver needs it:

To map registers:

```
reg_base = ioremap(mem->start, resource_size(mem))
```

MAC and DMA access requires these addresses.

2. struct cpmac_priv *priv

What it is used for:

This is the **private driver data**, unique to the cpmac driver.

Stored inside net_device using:

```
priv = netdev_priv(dev);
```

Contains **EVERYTHING** the driver needs at runtime:

- DMA ring pointers
- NAPI structure
- Locking
- PHY name
- Registers
- Pointers to buffers
- TX/RX stats

Typical members:

Member	Purpose
struct platform_device *pdev	Back reference to platform device
spinlock_t lock	Lock for TX/RX ops
spinlock_t rx_lock	Lock for RX path
struct napi_struct napi	NAPI poll handler
int ring_size	How many RX/TX descriptors
char phy_name[MII_BUS_ID_SIZE]	\${mdio_bus}:\${phy_id}
struct net_device *dev	Link back to netdev
void __iomem *regs	Mapped MAC registers
DMA descriptors	RX/TX descriptor arrays
buffer pointers	store real packet buffers
status counters	error counters

Why driver needs it:

The MAC + DMA engine cannot run without:

- descriptors
- RX/TX queues
- NAPI
- register mapping

All of these live inside cpmac_priv.

3. struct net_device *dev

What it is used for:

This is the **Linux representation of a network interface**, e.g. eth0.

This structure is what Linux networking stack uses.

Driver fills dev so Linux can:

- Send packets (ndo_start_xmit)
- Receive packets
- Set MAC address
- Open/close interface
- Query stats

Important members:

Member	Purpose
dev->netdev_ops	Function pointers for driver callbacks
dev->ethtool_ops	Provides settings to ethtool
dev->irq	Interrupt number
dev->dev_addr	MAC address
dev->mtu	Packet size (1500 default)
dev->name	"eth0", "eth1"
dev->flags	IFF_UP, IFF_RUNNING
dev->features	checksum offload, scatter-gather
dev->priv_flags	driver-specific flags

Why driver needs it:

- To register a network interface in Linux (register_netdev)
- To handle TX packets (ndo_start_xmit)
- To interact with NAPI
- To deliver packets to stack (netif_receive_skb)

This is the core interface between driver and OS.

4. struct plat_cpmac_data *pdata

What it is used for:

This contains **board-specific configuration** that cannot be auto-detected.

This includes:

- Which PHY to use
- MAC address stored in EEPROM
- PHY mask
- Port mode (MII/RMII/RGMII)

Members (typical):

Member	Purpose
unsigned char dev_addr[6]	MAC address
unsigned long phy_mask	Which PHYs are connected
int mac_index	Which MAC controller (0/1)
int full_duplex	board config

int speed	10/100
struct resource *dma_res	DMA resource
void (*reset)(void)	callback for reset

Why driver needs it:

- To choose correct PHY (phy_mask)
- To assign correct MAC address (dev_addr)
- To select correct MAC port instance
- To configure PHY mode

On boards like BeagleBone, this data comes from Device Tree.

5. struct phy_device *phydev

What it is used for:

Represents the **physical Ethernet transceiver** on the board.

PHY is responsible for:

- Auto-negotiation
- Speed (10/100)
- Duplex (half/full)
- Reading link status
- Converting electrical signals ↔ digital packets

Members (main ones):

Member	Purpose
int addr	PHY ID on MDIO bus
struct mii_bus *bus	MDIO bus controller
int speed	Current link speed
int duplex	Full/Half
int link	Up/Down
u32 supported	Capabilities
u32 advertising	What PHY advertises
phy_state	state machine
struct phy_driver *drv	actual PHY driver

Why driver needs it:

- To know link is up/down
- To know speed (100Mbps)
- To know duplex (full/half)
- To adjust MAC configuration
- To get interrupts for link changes

Driver must call:

`phy_connect()`

to hook MAC ↔ PHY together.

SUMMARY TABLE

Structure	Meaning	Key Purpose
struct resource	Hardware register/IRQ description	Tells MAC register addresses + interrupts
struct cpmac_priv	Driver private data	DMA rings, NAPI, locks, PHY name, descriptors
struct net_device	Linux interface (eth0)	Netdev ops, MAC addr, IRQ, MTU, features
struct plat_cpmac_data	Board config	PHY mask, MAC addr, mode (MII/RMII), etc.
struct phy_device	Physical PHY chip	Speed, link, duplex, MDIO bus

If you want next:

I can give you the **full internal structure layouts** (with all members), or explain how each structure is used inside:

- RX path
- TX path
- probe()
- open()
- start_xmit()
- poll()

Just tell me.

Okay. I will explain `alloc_skb()` like you are a complete beginner, in the simplest possible way, while still giving deep internal understanding.

No shortcuts. No assumptions. Very slow, very clear.

What is `alloc_skb()`? — Beginner Explanation

Think of `alloc_skb()` as:

 "Create a new empty packet for the Linux kernel."

Linux stores every packet (TX or RX) inside a **structure** called `sk_buff` (socket buffer).

So when Linux wants to create a packet, it calls:

`alloc_skb(size, gfp_flags);`

This function creates:

1. **The `sk_buff` structure** (the metadata container)
2. **A data buffer** (actual bytes of the packet)

Imagine a real-world analogy

 You want to send a letter.

You need:

3. **An envelope** → this is `struct sk_buff`
4. **A blank paper inside** → this is the data buffer

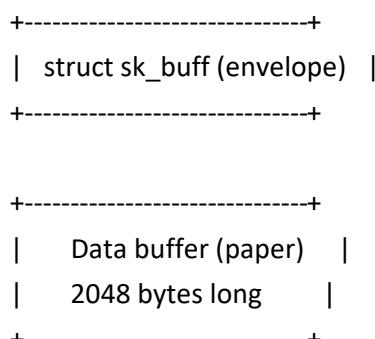
`alloc_skb()` gives you **both**.

What exactly is allocated?

When you call:

`skb = alloc_skb(2048, GFP_KERNEL);`

Linux allocates:



These two are separate memory allocations.

What does `sk_buff` contain? (beginner-friendly)

The `sk_buff` is HUGE and powerful.

But as a beginner, remember just 6 important fields:

Field	Meaning
head	start of the data buffer
data	pointer where valid data begins
tail	pointer where valid data ends
end	end of total buffer
len	amount of valid data in packet
dev	which network interface (eth0)

After alloc_skb(), the state is:

```
head = data = tail -----> empty area -----> end
len = 0
```

📘 Why alloc_skb() does NOT add any data?

Because alloc_skb() only gives you:

- the empty envelope
- the blank paper

YOU have to write the data later using:

```
skb_put()
skb_push()
```

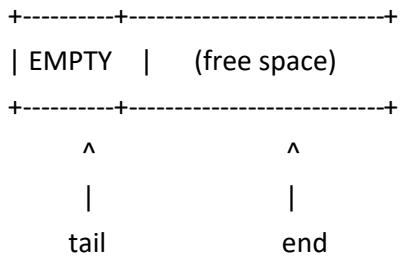
💻 Detailed Memory Layout — VISUAL for Beginners

After alloc_skb(2048):

```
head==data==tail
```

|

v



len = 0

This means:

- No data yet
- You must *grow* the packet with functions

🧠 Key Helper Functions (Very Important for beginners)

★ skb_put(skb, length)

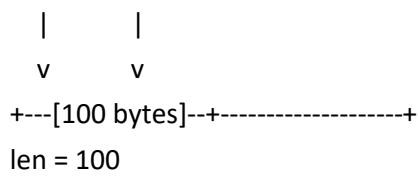
Adds data at the **end**.

Example:

```
char *ptr = skb_put(skb, 100);
memcpy(ptr, mydata, 100);
```

Layout becomes:

head=data tail



★ skb_push(skb, len)

Adds space at the **front** (for headers)

Used to add Ethernet/IP/TCP headers.

★ skb_reserve(skb, len)

Moves data and tail forward without adding actual data.

(This is used to leave headroom.)

THE REAL DIFFERENCE (In One Line)

Function	What it does	When used
skb_reserve()	Moves head + data pointers forward , creating empty space at the beginning of a newly allocated skb. Does not add data .	Used right after allocating skb , to reserve space for headers the driver will add later.
skb_push()	Moves data pointer backward , making room to add actual header bytes , and returns pointer where you can write header data.	Used when constructing a packet , to <i>actually add headers</i> (Ethernet/IP/TCP).

🧠 THINK LIKE THIS:

✓ skb_reserve() = “Please leave some empty space at the beginning; I will use it later.”

✓ skb_push() = “I am now writing a header in that space.”

🛠 Why Drivers Use alloc_skb()

Ethernet drivers use it for:

TX Path

Upper layers (TCP/IP) allocate skb → driver reads skb → sends to hardware.

RX Path

Driver prepares skb buffers → hardware fills them → driver sends the skb to kernel.

👉 Simplest Driver Flow With alloc_skb() (Beginner Friendly)

✓ TX (Sending data)

5. Application sends data

6. Kernel builds packet

7. Kernel calls:

```
skb = alloc_skb(2048, GFP_KERNEL);
skb_put(skb, data_length);
```

8. Driver takes skb → gives it to hardware

9. Hardware sends to wire

✓ RX (Receiving data)

10. Driver prepares empty skb:
`skb = netdev_alloc_skb(dev, 2048);`
11. Hardware receives packet into this buffer
12. Driver updates skb:
`skb_put(skb, actual_packet_length);`
13. Driver sends skb up using:
`netif_rx(skb);`
14. Kernel processes it (IP → TCP → app)

 **Why two different functions?**

`alloc_skb()`

Used by network protocols (TCP, IP)

`dev_alloc_skb() / netdev_alloc_skb()`

Used by network drivers

They add extra alignment and headroom for Ethernet headers.

Because these APIs:

Ensure correct `NET_IP_ALIGN` (2-byte alignment)

NICs require that IP headers are aligned to 2 bytes for performance.

Ensure DMA-safe memory

Some NICs cannot DMA from any random memory.

Provide proper skb headroom for Ethernet header

Are optimized for fast-path RX in drivers

 **Very Important for drivers**

Drivers must use:

`GFP_ATOMIC`

Because they run in:

- interrupt context
- NAPI context

Where sleeping is not allowed.

 **What you must remember as a beginner**

`alloc_skb(size)` creates an empty packet

You must fill it with data using `skb_put()`

Drivers use `netdev_alloc_skb()`

RX path uses `GFP_ATOMIC`

`sk_buff` is the most important network structure in the kernel

 **If you want, I can explain next:**

full `sk_buff` memory blocks (head, data, tail, end)

- detailed diagrams for push, pull, reserve, put
- how alloc_skb works inside NAPI driver
- what happens from wire → DMA → skb → kernel → tcpdump
- how to instrument / log skb inside Beaglebone Black driver

Just tell me "**explain next**" or what part you want.

Great — now I will explain the **skb destructor pointer** in **very simple beginner language**, but with **deep internal understanding**.

You will understand **what it is, why it exists, when it is called, and what sock_wfree/sock_rfree do**.

Let's go **slow** and **clear**.

WHAT IS skb->destructor? (Beginner Explanation)

destructor is a **function pointer** inside struct sk_buff.

Think of it as:

 **"A callback function that is called automatically when the skb is destroyed."**

In simple words:

When the kernel finishes using the skb and frees it,

it will automatically call skb->destructor() **if it is set**.

This allows the owner (socket, TCP, driver) to do **cleanup work**.

When does sk_buff get destroyed?

Whenever:

- Packet is **transmitted**, and driver frees it
- Packet is **dropped**
- Packet is **consumed** by upper layers
- Socket finishes using it

At that moment, the skb memory will be freed.

Before freeing memory, Linux checks:

```
if (skb->destructor)
    skb->destructor(skb);
```

So **destructor is the last thing** called before skb deletion.

WHY do we need a destructor?

Because sometimes **someone owns this skb**, and they need to do something when the skb disappears.

For example:

 A socket may want to:

- Update memory usage counters
- Release some resources
- Notify TCP that memory is free
- Adjust internal accounting

 A driver may want to:

- Unmap a DMA buffer
- Free metadata
- Decrement reference counts

⌚ What happens if skb has NO destructor?

If skb->destructor = NULL, then:

⌚ skb is freed normally

⌚ No extra work is done

This is true for **most driver RX skbs**, because they do not belong to a socket.

⌚ Who sets the destructor?

✓ If skb comes **from driver RX path**

→ No socket involved

→ No need to track memory

→ destructor = NULL

✓ If skb belongs to a **socket (TCP/UDP)**

→ Kernel sets destructor using:

skb_set_owner_r(skb, sk);

skb_set_owner_w(skb, sk);

These functions will set:

- sock_rfree or

- sock_wfree

as the destructor.

屏障 Two most common destructors

1. sock_rfree

Used when the packet belongs to a socket **receive** queue.

Meaning:

- App receives a packet
- skb stored in socket receive queue
- After app consumes it → skb freed
- destructor runs → sock_rfree()

This tells socket:

⌚ “Hey, this skb is gone, reduce memory usage.”

2. sock_wfree

Used for **transmit** side.

Meaning:

- App sends a packet
- skb queued for transmission
- After NIC actually sends it → skb freed
- destructor runs → sock_wfree()

This tells socket:

⌚ “This sent packet’s memory is freed, you can send more.”

This is CRITICAL for TCP congestion control and memory accounting.

💻 WHAT HAPPENS INSIDE sock_wfree?

Very important for understanding TCP.

`sock_wfree()`:

- releases memory from socket memory accounting
- wakes up processes waiting for send buffer space
- important for TCP flow control

Without this:

- TCP send buffer would never reduce
- The application would block forever on `send()`
- You could not send new packets

Real example: What if destructor is wrong?

If the driver mistakenly overwrites `skb->destructor`:

You get kernel warnings like:

`skb_release_data: Warning: destructor is still set!`

Or even memory leaks or crashes.

Example scenario for beginners

Application sends packet:

`send()`

Kernel creates skb:

```
skb = alloc_skb()  
skb_put(skb, data_length)  
skb_set_owner_w(skb, socket)
```

→ This sets:

`skb->destructor = sock_wfree`

Driver sends packet

Packet transmitted

Driver frees skb

Kernel calls:

`sock_wfree(skb)`

Socket adjusts memory usage.

When does DRIVER need a destructor?

RX Path → **NO**, because skb is handed to kernel

TX Path → usually **NO**, unless:

- the driver uses special DMA tricks
- wants to clean private data when freeing skb
- wants custom callbacks after send

Most drivers **do not set** a destructor.

Summary for Beginners

- `skb->destructor` = function called before skb is freed
- Used mostly for **socket accounting**
- Set by `skb_set_owner_r()` or `skb_set_owner_w()`
- Most drivers do **NOT** touch it
- Common destructors: `sock_rfree` & `sock_wfree`
- Important for TCP memory & flow control

If you want, I can explain next:

✓ The entire `sk_buff` memory layout (`head`, `data`, `tail`, `end`)

- ✓ What **headroom** and **tailroom** actually mean
- ✓ How `skb_put`/`push`/`pull`/`reserve` work with diagrams
- ✓ How a driver handles skb in RX/TX path
- ✓ What happens to skb in **NAPI**
- ✓ How skb finally gets freed in Linux

Just tell me: "**explain headroom/tailroom**", or ask anything next.

Below is a **VERY BEGINNER-FRIENDLY, step-by-step, dead-simple** explanation of:

- What is `struct timer_list`?**
- Why do network drivers use it?**
- What are the important timer functions?**
- How timers actually work inside the Linux kernel?**

1. What is `struct timer_list`?

`struct timer_list` is a **kernel timer** object.

You can think of it like:

an alarm clock inside the Linux kernel

You give the timer:

- a **callback function** (function to run when time expires)
- an **expiry time** (when to run)
- some **data to pass** to that function

The kernel will wake up and run your callback function **in interrupt context** when the time comes.

2. Why do network drivers use timers?

Real network drivers get hardware interrupts like:

- **RX interrupt** → “packet received”
- **TX interrupt** → “packet transmitted”

But in a **pseudo driver**, or when hardware is slow, or when hardware has no real interrupts, we use a **timer** to:

- poll hardware registers
- simulate a packet arriving every second
- implement link check (carrier detection)
- clean TX buffers regularly
- retry operations

So timers help network drivers **simulate or implement periodic work**.

Example:

Every 1 second → check if a fake packet should be generated → deliver it.

This is exactly what we used in your pseudo driver.

3. What does struct timer_list look like internally?

You don't need full details, but simplified:

```
struct timer_list {  
    unsigned long expires;    // when to run  
    void (*function)(struct timer_list *); // callback  
    struct list_head entry;   // kernel internal use  
    unsigned int flags;      // kernel internal  
    ...  
};
```

4. How do you create and use a timer?

Step 1 — Declare a timer in your private driver data:

```
struct pseudo_priv {  
    struct timer_list rx_timer;  
};
```

Step 2 — Initialize the timer (in open())

You use:

```
timer_setup(&priv->rx_timer, callback_func, 0);
```

This assigns:

- `callback_func` → will be called when timer fires.

Step 3 — Start the timer

```
mod_timer(&priv->rx_timer, jiffies + HZ);
```

Meaning:

- `jiffies` = current kernel ticks
- `HZ` = 1 second
- so call `callback_func` **1 second later**

5. VERY IMPORTANT: Timer callback function

Example:

```
static void pseudo_rx_timer(struct timer_list *t)  
{  
    // 1) Get pointer to private struct  
    struct pseudo_priv *priv = from_timer(priv, t, rx_timer);  
  
    // 2) Generate fake packet  
    // 3) Send it to network stack  
    // 4) Re-arm timer for next time
```

```
    mod_timer(&priv->rx_timer, jiffies + HZ);
}
```

So the callback:

- Is like an interrupt handler
- Must NOT sleep
- Must finish quickly
- Is used to do scheduled work (like packet generation)

6. Why use from_timer()?

Because inside the callback, you only receive a pointer to the timer:

t → struct timer_list*

But you want access to your driver private data priv.

To get it, use:

```
struct pseudo_priv *priv = from_timer(priv, t, rx_timer);
```

This tells the kernel:

"Give me the struct that contains this timer as a member."

7. How do timers behave in the kernel? (Important)

Timelines:

```
timer_setup(...)    <-- initializes the timer  
mod_timer(..., X)  <-- schedules it to expire at time X
```

Time passes...

Kernel timer subsystem wakes up,
checks expired timers,
calls your callback function.

Your callback executes,
and (if needed) re-schedules itself via mod_timer()

The callback runs in **softirq context**, NOT in a normal process.

This means:

✗ You CANNOT sleep inside callback

✗ You CANNOT block

✓ You CAN allocate skbs

✓ You CAN deliver packets

✓ You CAN update net_device stats

✓ You CAN restart timer

8. Why do we put timer setup in ndo_open() and not in init()?

Because:

- Inside init(), the network device is **not yet registered**
- Its data structures might not be fully initialized
- Kernel may call your callback before device is ready
→ **crash or segmentation fault**

So correct sequence:

init() → alloc netdev
→ register netdev

open() → setup timer
→ start queue

This prevents segmentation faults.

9. Summary in beginner language

Concept	Meaning
timer_list	A built-in kernel alarm clock
timer_setup()	Register what function to run when timer fires
mod_timer()	Start / re-start the timer
Callback	Function executed automatically when timer expires
Used for	Simulating packets, polling hardware, periodic checks
Runs in	Softirq (like interrupt), must be fast

10. If you want:

I can also give you:

- A drawing diagram of how timers work
- Flowchart of timer lifecycle
- A super-small example program using only one timer
- Explain how real NIC drivers use timers vs interrupts

Just tell me.

TX path

Wednesday, November 26, 2025 4:49 PM

Below is a **complete, deep, theoretical RX (receive) path explanation** for a typical Linux Ethernet driver (including how it applies to **CPMAC / BeagleBone-style drivers**).

I will explain:

1. **What happens in hardware (MAC + DMA + descriptors)**
2. **What happens in the kernel (NAPI + net_device + driver callbacks)**
3. **How your driver structures (priv, resource, phy_device, etc.) fit into the RX path**
4. **Step-by-step flow of ONE packet from wire → kernel socket**

1. HIGH-LEVEL RX FLOW

Packet comes from wire → PHY → MAC → DMA → Driver → Kernel stack → User application
(Network Cable)



PHY → Converts signals to digital bits



MAC → Understands Ethernet, CRC, frame boundaries



RX DMA Engine → Writes incoming packet into memory buffer



Driver RX Interrupt/NAPI → Driver reads packets into skb



▼
Linux Network Stack → IP → TCP/UDP

|

▼

Socket Buffer → Application (recv(), etc.)

2. RX PATH IN REAL DRIVERS (like CPMAC on BeagleBone)

Step 1: Driver allocated RX Ring (DMA descriptors)

During driver init (probe()), the driver allocates:

RX Descriptor Ring (priv->desc_ring or priv->rx_head etc.)

- Array of descriptors in RAM
- Each descriptor points to an RX buffer
- Hardware writes into these buffers

Descriptor contains:

```
struct dma_desc {  
    u32 buffer_addr; // Physical address of RX buffer  
    u32 length;     // Length written by hardware  
    u32 status;     // OWN bit, errors, etc.  
}
```

Step 2: Hardware (MAC) receives packet from PHY

- PHY gets bits from cable
- Converts to digital symbols
- Sends to MAC through **MII/RMII** interface
- MAC decodes frame, checks FCS
- Hands packet to DMA
- DMA writes it directly into RX buffer
- Sets descriptor OWN bit = DRIVER (ready to read)

Step 3: Hardware triggers RX interrupt

Interrupt name from device tree: "irq"

The driver registered:

```
dev->irq = platform_get_irq_byname(pdev, "irq");
```

Hardware fires interrupt when:

- Packet received
- DMA finished writing
- Error

Step 4: Driver interrupt handler schedules NAPI

Interrupt handler **never processes packets directly.**

It calls:

```
napi_schedule(&priv->napi);
```

This marks the device ready for polling.

! NAPI RX Path (Very important)

NAPI reduces interrupts by polling multiple packets.

Driver registered NAPI in probe:

```
netif_napi_add(dev, &priv->napi, cpmac_poll, 64);
```

Meaning:

- Poll function = cpmac_poll()
- Weight = 64 packets per poll loop

Step 5: cpmac_poll() runs → fetches packets

Inside cpmac_poll():

5. Loop over RX descriptors
6. Check descriptor status:
 - OWN = DRIVER? (means new packet)
7. If packet present:
 - Read length

- Copy pointer to skb
 - Give packet to kernel using:
`netif_receive_skb(skb);`
8. Refill the descriptor with new buffer
 9. Set OWN = HARDWARE again

This is the heart of RX.

Step 6: Kernel Network Stack processes skb

Packet enters stack:

```
netif_receive_skb()
  → __netif_receive_skb_core()
    → IP layer
    → TCP/UDP layer
    → Socket buffer queue
```

Step 7: User receives packet

Application calls:

`recv()`, `recvfrom()`, `read()`, etc.

It finally gets the packet payload.

NOW RELATE EVERYTHING TO THE STRUCTURES THE DRIVER USED

**struct resource mem*

Represents the **MAC register memory region**.

Used for RX:

- Driver reads RX DMA status registers
- Driver configures RX enable
- Driver programs RX descriptor base address

Example hardware registers it points to:

RX_CONTROL_REG
RX_DESC_BASE_REG
RX_INT_STATUS_REG
RX_INT_ENABLE_REG
MAC_STATUS_REG

**struct cpmac_priv priv*

Driver private structure for one Ethernet interface.

Important RX members:

```
struct napi_struct napi;      // NAPI polling structure for RX
spinlock_t rx_lock;          // Protect RX ring
int ring_size;               // Number of RX descriptors
struct dma_desc *rx_head;    // Pointer to RX descriptor ring
struct sk_buff **rx_buf;     // Array of RX buffer pointers
struct net_device *dev;       // Back pointer to network device
```

Used for RX path:

- RX DMA descriptor ring management
- Buffers
- Locking
- NAPI

**struct net_device dev*

Represents this network interface to Linux.

For RX:

- dev->netdev_ops->ndo_open() starts RX DMA
- dev->netdev_ops->ndo_stop() stops it
- MAC address (dev->dev_addr) used to filter RX frames

Notable fields:

```
unsigned char dev_addr[6]; // MAC address
int irq; // RX interrupt line
netdev_ops // contains ndo_open, ndo_start_xmit
ethtool_ops
```

**struct plat_cpmac_data pdata*

Platform-specific board info.

For RX:

- Contains PHY mask → determines which PHY addresses to probe
- Contains MAC address (dev_addr)
- Contains special flags for RX/TX behavior

Ex:

```
struct plat_cpmac_data {
    u8 dev_addr[6];
    u32 phy_mask; // which PHYs exist → determines RX path
    int rmii_en; // RMII or MII mode
};
```

**struct phy_device phydev*

Represents the PHY chip.

Used for RX path:

- Link status (up/down)
- Negotiated speed (10/100)
- Duplex
- Whether RX lanes are active

Driver connects via:

```
phydev = phy_connect(dev, priv->phy_name, cpmac_adjust_link,
PHY_INTERFACE_MODE_MII);
```

PHY sends:

- RX activity
- Speed changes
- Duplex changes
- Link lost

The driver configures MAC RX settings based on PHY signals.

★ COMPLETE LIVE RX EXAMPLE (REALISTIC)

Scenario:

A PC sends:

"HELLO" UDP packet to 192.168.1.10 (BeagleBone)

Step-by-step:

1 — PHY receives electrical signal

- Bits from cable → decoded
- PHY sends digital MII frames to MAC

2 — MAC receives Ethernet frame

MAC decodes:

Dest MAC = BeagleBone MAC

Payload = IP + UDP + "HELLO"

Length = 42 bytes

3 — DMA writes to RX buffer

- Using descriptor 5 in RX ring
- Writes frame into buffer
- Fills descriptor.length = 42
- Clears OWN bit → DRIVER owns descriptor now

4 — Hardware raises RX interrupt

Interrupt fires on "irq" line

Driver interrupt handler wakes NAPI:
napi_schedule(&priv->napi);

5 — cpmac_poll() runs

Inside poll:

```
desc = rx_ring[5]
length = desc->length = 42
skb = build_skb(desc->buffer)
netif_receive_skb(skb); ← send to kernel
refill descriptor with new buffer
set OWN = HARDWARE
```

6 — Linux network stack

Processing:

Ethernet → IP → UDP

UDP demux sees destination port matches application.

7 — User application receives

Application calling:

```
recvfrom(sock, buffer, ...)
```

gets:

"HELLO"

Conclusion

This is the **complete theoretical RX path**, with:

- Hardware perspective
- Driver perspective

- Kernel networking perspective
- How each structure (priv, phy_device, net_device, resource, pdata) is used

If you want, I can also give:

- TX path in the same detailed way
- Draw RX descriptor ring diagrams
- Expand every function in cpmac_probe() step-by-step in RX context
- Give full "packet lifetime" logs with printk

Just tell me "**Give TX path**" or "**Give packet lifetime example**".

Code review

Friday, November 28, 2025 2:38 PM

Below is a **simple, beginner-friendly, dead-detail explanation** of the entire pseudo network driver you posted.

I will explain **each function, each struct, and the complete flow step-by-step** so you understand *why* everything exists and *how* it works.

BIG PICTURE: WHAT THIS DRIVER DOES

This is a **fake (pseudo) network driver**, meaning:

- It creates a fake network interface like → **pnet0**
- TX path (sending) → it pretends to send packets but actually just drops them
- RX path (receiving) → it generates fake packets every 1 second using a **kernel timer**
- It updates statistics like a real driver
- No real hardware exists

You get something like this in Linux:

```
$ ifconfig pnet0 up  
$ tcpdump -i pnet0 # you see fake packets
```

=====

STRUCT pseudo_priv

```
struct pseudo_priv {  
    struct net_device *ndev;  
    struct timer_list rx_timer;  
};
```

Purpose:

This is the **private data of your network driver**.

Every network driver usually maintains:

- software state
- timer/event handles
- custom buffers
- statistics
- TX/RX rings

Your driver only needs 2 things:

Field	Meaning
ndev	Pointer to the net_device structure. Needed so the timer can know which network device the packet belongs to.
rx_time	A timer_list used to generate fake packets every 1 second.

```
=====
```

NETDEVICE OPS TABLE

```
static const struct net_device_ops pseudo_netdev_ops = {  
    .ndo_open     = pseudo_open,  
    .ndo_stop     = pseudo_stop,  
    .ndo_start_xmit = pseudo_start_xmit,  
};
```

This table tells the Linux networking stack:

“When someone opens the device → call pseudo_open
when someone closes → call pseudo_stop
when someone transmits a packet → call pseudo_start_xmit.”

These functions implement your driver behavior.

```
=====
```

FUNCTION EXPLANATION (Dead Simple)

1 pseudo_start_xmit() — TX PATH

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

Called when:

ping pnet0
curl <http://...>

TX path logic:

1. Update statistics
2. Print log
3. Drop/free the packet (fake send)
4. Tell Linux “OK done”

Breakdown:

```
dev->stats.tx_packets++;  
dev->stats.tx_bytes += skb->len;
```

Counts packets and bytes sent.

```
dev_kfree_skb(skb);
```

Drops packet (because there is no real hardware).

Returns:

```
NETDEV_TX_OK
```

→ means transmission "successful".

2 pseudo_rx_timer() — RX PATH (Timer Callback)

This function is called **EVERY 1 second** by the kernel timer.

Purpose:

- ✓ Create a fake incoming Ethernet packet
- ✓ Fill it with dummy data
- ✓ Deliver it to the upper layers (IP stack)
- ✓ Update statistics
- ✓ Restart the timer

Step-by-step:

Step 1: Get private data

```
struct pseudo_priv *priv = from_timer(priv, t, rx_timer);
```

Step 2: Allocate skb

```
skb = netdev_alloc_skb(dev, pkt_len + NET_IP_ALIGN);
```

This allocates a buffer for the fake packet.

Step 3: Fill buffer with dummy data

```
skb_put(skb, pkt_len);
memset(skb->data, 0xAB, pkt_len);
```

Packet content is 0xAB repeated.

Step 4: Set protocol (Ethernet type)

```
skb->protocol = eth_type_trans(skb, dev);
```

Step 5: Update stats

```
dev->stats.rx_packets++;
dev->stats.rx_bytes += pkt_len;
```

Step 6: Deliver packet to network stack

```
netif_rx(skb);
```

Now Linux thinks a real packet arrived.

Step 7: Restart timer

```
mod_timer(&priv->rx_timer, jiffies + HZ);
```

→ 1 second later, more fake packets will come.

3 pseudo_open() — Device UP

Called when you run:

```
sudo ifconfig pnet0 up
```

Actions:

5. Print “open”
6. Setup timer
7. Start network queue
`timer_setup(&priv->rx_timer, pseudo_rx_timer, 0);
mod_timer(&priv->rx_timer, jiffies + HZ);`

→ Start the receive timer (fake packets every second)

4 `pseudo_stop()` — Device DOWN

Called when you run:

```
sudo ifconfig pnet0 down
```

Actions:

8. Stop the TX queue
9. Cancel timer safely
`del_timer_sync(&priv->rx_timer);`

5 `pseudo_setup()` — Device Initialization

Called *automatically* by `alloc_netdev()`.

Purpose:

- ✓ setup as Ethernet device
- ✓ assign MAC address
- ✓ define MTU
- ✓ add ops

Key part:

```
u8 fake_mac[ETH_ALEN] = {0x02,0x00,0x00,0x00,0x00,0x01};  
eth_hw_addr_set(dev, fake_mac);
```

This sets MAC address safely (because `dev_addr[]` is read-only now).

6 `pseudo_init()` — Module Loading

Executed when you run:

```
sudo insmod simple_pseudo.ko
```

Actions:

10. Allocate one network device
11. Fill private data
12. Register with kernel
13. Print success

7 `pseudo_exit()` — Module Removal

Executed when you run:

```
sudo rmmod simple_pseudo
```

Actions:

14. Unregister interface pnet0
15. Free memory

```
=====
```

WHAT HAPPENS IN REAL LIFE FLOW

When the module loads:

```
insmod simple_pseudo.ko
```

Flow:

```
pseudo_init()  
    alloc_netdev()  
        pseudo_setup()  
    register_netdev()
```

Linux now creates:

```
pnet0
```

When you bring it UP:

```
ifconfig pnet0 up
```

Flow:

```
ndo_open → pseudo_open  
    start timer  
    start queue
```

Every 1 second:

```
pseudo_rx_timer()  
    generate skb  
    deliver fake packet
```

When you do TX:

```
ping -I pnet0 1.1.1.1
```

```
-----
```

```
pseudo_start_xmit()  
    drop packet  
    print
```

When DOWN:

```
ifconfig pnet0 down
```

```
pseudo_stop()  
    stop timer  
    stop queue
```

When module removed:

```
rmmod simple_pseudo
```

Flow:

```
pseudo_exit()  
    unregister  
    free memory
```

If you want next:

- ✓ Add ethtool support
- ✓ Add real TX/RX queues
- ✓ Add NAPI version
- ✓ Add sysfs entries
- ✓ Add debugfs

Just tell me.

Steps to build

Tuesday, November 18, 2025 1:21 PM

Got it — you want **the step-by-step checklist** for writing an Ethernet driver, similar to how we explain a simple character driver (allocate major/minor → create cdev → create class → open/read/write, etc.).

Below is **theoretical but practical**, exactly the order a real Linux Ethernet driver is written.

If you follow this checklist, you can map each step to real drivers like **TI CPSW, e1000, r8169, mlx5**, etc.

“How to Write an Ethernet Driver” — Step-By-Step Like a Character Driver Guide

Below is the *correct order* of everything you must create in an Ethernet driver — from structs → registration → ring buffers → NAPI → TX/RX → IRQ → shutdown.

I will keep it **conceptual**, NO heavy code.

STEP 1 — Include Necessary Kernel Headers

Like char driver includes <linux/fs.h>, net drivers include:

- <linux/netdevice.h>
- <linux/etherdevice.h>
- <linux/ethtool.h>
- <linux/skbuff.h>
- <linux/interrupt.h>
- <linux/napi.h>

→ These give you access to struct `net_device`, `napi_struct`, RX/TX functions, skb API, etc.

STEP 2 — Create struct `net_device`

This is equivalent to **allocating cdev in character driver**.

Use:

```
alloc_etherdev(sizeof(private_struct));
```

This allocates:

- `net_device` structure (kernel interface)
- private memory for your driver (like `file_operations` private data)

Inside `net_device`, important fields you will fill later:

- `.netdev_ops` → send packet, open, stop
- `.ethtool_ops` → speed, stats, settings
- `.mtu`
- `.dev_addr` → MAC address

STEP 3 — Create Your Driver’s Private Structure

Like `mydev_t` in char driver.

Example fields inside private struct:

```

struct myeth_priv {
    void __iomem *regs; // mapped hardware registers
    struct napi_struct napi; // NAPI context
    struct net_device *ndev; // back pointer
    struct ring rx_ring; // RX ring descriptors
    struct ring tx_ring; // TX ring descriptors
    spinlock_t lock; // protect TX
    int irq; // IRQ number
};


```

This private data holds everything hardware-specific.

STEP 4 — Implement `net_device_ops`

Equivalent to char driver's `file_operations`.

```

static const struct net_device_ops my_netdev_ops = {
    .ndo_open      = my_open,
    .ndo_stop     = my_stop,
    .ndo_start_xmit = my_xmit,
    .ndo_set_mac_address = my_set_mac,
    .ndo_get_stats64 = my_get_stats,
};


```

You will register this in:

```
ndev->netdev_ops = &my_netdev_ops;
```

STEP 5 — Probe/Init Hardware (Platform or PCI Probe)

This is like "device detection" step.

Inside your probe:

1. Map I/O registers (`ioremap`)
2. Request IRQ
3. Read MAC from hardware
4. Allocate descriptor rings
5. Setup NAPI
6. Register `net_device` with kernel:
`register_netdev(ndev);`

This makes the driver appear as `eth0`.

STEP 6 — Create TX Ring Buffer

TX ring contains descriptors telling hardware where packets are.

Each TX descriptor contains:

- buffer address
- buffer length
- flags

Driver steps:

7. Allocate TX buffers (DMA coherent)

8. Fill each descriptor with buffer info

9. Mark descriptors as "free"

Like:

```
priv->tx_ring.desc[i].buffer = dma_addr;
```

```
priv->tx_ring.desc[i].status = FREE;
```

STEP 7 — Create RX Ring Buffer

RX ring stores empty buffers for hardware to DMA incoming packets into.

RX descriptor contains:

- buffer address (skb->data)
- buffer size
- "ownership" flag (HW or SW)

Steps:

10. Allocate SKBs for RX ring using netdev_alloc_skb

11. Map skb->data for DMA with dma_map_single

12. Give descriptors to hardware (OWN flag)

STEP 8 — Implement ndo_start_xmit() (TX Path)

This is like char driver's .write function.

TX flow inside driver:

13. Called by kernel when upper layers want to send packet
14. Driver picks free TX descriptor
15. dma_map the skb
16. Fill descriptor
17. Tell hardware to start TX
18. Free descriptor on TX-complete interrupt

STEP 9 — Request IRQ Handler

Like char driver: request_irq()

Interrupt sources:

- RX complete
- TX complete

- Link change
- ISR Responsibilities:
- Disable future interrupts
 - Schedule NAPI polling
`napi_schedule(&priv->napi);`

STEP 10 — NAPI Poll Function (RX Path)

This is the heart of RX.

NAPI poll does:

19. Check RX descriptors for received packets
20. Turn each descriptor into an skb
21. Push packet into networking stack →
`netif_receive_skb(skb);`

22. Refill RX descriptors with new SKBs
23. Return number of packets processed

This replaces interrupt-driven RX (faster!)

STEP 11 — XDP (Optional Advanced Path)

XDP adds a **bypass path** BEFORE skb creation.

- Runs in driver
- No SKB
- No kernel networking stack
- Zero-copy using "umem"

XDP allows:

- Drop
- Redirect
- Modify packet
- Forward to another NIC

You add XDP support using:

`ndo_bpf`

`ndo_xdp_xmit`

STEP 12 — Implement Stop / Remove

When interface goes down:

- disable interrupts
- free DMA rings
- unmap memory
- unregister device

STEP 13 — Module Init & Exit

Like char driver:

```
module_init(my_init);  
module_exit(my_exit);
```

Full Summary (High-Level Roadmap)

Step	What You Create	Why
1	Headers	Required for net stack
2	alloc_etherdev()	Create net_device
3	Private struct	Store driver state
4	netdev_ops	Open/Close/Transmit
5	Probe	Detect hardware
6	TX ring	Hardware sends packets
7	RX ring	Hardware receives packets
8	xmit()	TX handler
9	IRQ	RX/TX interrupt
10	NAPI poll	Fast RX processing
11	(optional) XDP	Fast-path packet processing
12	stop/remove	Cleanup
13	module_init/exit	Driver entry points

Perfect — you want a full deep-dive into every structure and function involved in writing a Linux Ethernet driver.

This is the "must-know API set" for ANY network driver:

- Core net_device structures
- NAPI structures
- DMA descriptor structures
- SKB-related structures
- Operations tables (netdev_ops & ethtool_ops)
- Driver internal structures

I will explain each one, field-by-field, and show why it exists, when it is used, and who calls it.

1. struct net_device

This is the main object representing a network interface (eth0, eth1, etc).

Allocated using:

```
struct net_device *ndev = alloc_etherdev(sizeof(struct my_priv));
```

Important fields inside struct net_device:

Field	Purpose
name	Interface name like "ethX"

dev_addr[6]	MAC address
netdev_ops	Methods your driver provides (open, stop, transmit, etc.)
ethtool_ops	Settings: speed, link modes, stats
mtu	Max packet size
private area	Private driver struct (your state)

★ 2. struct net_device_ops

Equivalent of char driver's file_operations.

You fill this with function pointers:

```
static const struct net_device_ops my_netdev_ops = {
    .ndo_open      = my_open,
    .ndo_stop      = my_stop,
    .ndo_start_xmit = my_xmit,
    .ndo_set_mac_address = my_set_mac,
    .ndo_change_mtu   = my_change_mtu,
    .ndo_get_stats64  = my_get_stats,
};
```

Important ops:
`ndo_open()`

Called when user does:

`ip link set eth0 up`

You:

- enable hardware
 - allocate DMA rings
 - enable interrupts
 - enable NAPI
- `ndo_stop()`

Called when:

`ip link set eth0 down`

You:

- stop HW
- free rings
- disable interrupts
- disable NAPI

`**ndo_start_xmit(struct sk_buff skb, struct net_device ndev)`

Called when kernel wants to transmit a packet.

You must:

- take packet data from skb
- map to DMA
- fill TX descriptor

- push to HW
`ndo_get_stats64()`
- Return RX/TX packet counts.

★ 3. struct net_device_priv (Your Driver Structure)

This is your own structure stored inside `alloc_etherdev()` private memory.

Example fields you put:

```
struct my_priv {
    struct napi_struct napi;
    struct net_device *ndev;

    void __iomem *hw_regs;

    struct my_ring rx_ring;
    struct my_ring tx_ring;

    spinlock_t tx_lock;

    int irq;
    u64 stats_rx;
    u64 stats_tx;
};
```

This structure stores **everything private to your hardware**.

★ 4. NAPI: struct napi_struct

NAPI improves RX performance by polling instead of interrupt storm.

Allocated via:

```
netif_napi_add(ndev, &priv->napi, my_poll, 64);
```

Important fields:

Field	Purpose
<code>poll()</code>	Driver function to process RX packets
<code>weight</code>	Max packets processed per poll
<code>state</code>	whether NAPI is scheduled

NAPI Flow:

24. IRQ fires (RX interrupt)
25. Driver disables IRQ
26. Calls `napi_schedule()`
27. Kernel calls driver `poll()`
28. Driver processes RX packets

29. When no more packets → calls `napi_complete_done()`
30. Driver re-enables IRQ

★ 5. DMA Descriptor Rings (RX & TX Rings)

Your NIC hardware uses rings to store packet metadata.

Example (generic):

```
struct my_desc {
    dma_addr_t buffer;
    u32 length;
    u32 flags;
};
```

TX Ring Fields:

Field	Meaning
buffer	Where packet is stored
length	Packet length
flags	OWN bit, FIRST, LAST

RX Ring Fields:

Field	Meaning
buffer	Empty buffer for HW to fill
length	Max buffer size
status	OWN bit, Filled bit

★ 6. struct sk_buff (SKB)

This is the most important packet structure in Linux.

When receiving:

```
struct sk_buff *skb = build_skb(rx_buffer, size);
```

SKB represents:

- packet data
- metadata
- protocol fields
- timestamps
- checksum info
- socket association

Important SKB fields:

Field	Meaning
data	Pointer to packet data
len	Packet length
head	Start of allocated buffer
dev	Which NIC received it
protocol	EtherType
ip_summed	Whether HW checksum was done

You pass skb to network stack:

```
netif_receive_skb(skb);
```

For TX:

- Kernel builds SKB
- Calls your `ndo_start_xmit()`

★ 7. Interrupt Handler

Registered using:

```
request_irq(priv->irq, my_isr, 0, "myeth", priv);
```

ISR job:

- Read interrupt status registers
- Disable interrupts
- Schedule NAPI:
`napi_schedule(&priv->napi);`

Why disable interrupts?

→ Let NAPI poll handle RX.

★ 8. XDP (Extra Data Path) Structures

If you support XDP, you'll use:

```
struct xdp_rxq_info
```

Describes RX queue for XDP.

```
struct xdp_buff
```

Represents raw packet:

Field	Meaning
data	start of packet
data_end	packet boundary
data_hard_start	headroom
rxq	which queue

XDP-supporting NICs attach program via:

`ndo_bpf`

`ndo_xdp_xmit`

XDP is optional.

★ 9. ethtool operations: struct ethtool_ops

Used for commands like:

- `ethtool -s eth0 speed 100 duplex full`
- `ethtool -i eth0`
- `ethtool -k eth0`

Driver fills:

```

static const struct ethtool_ops my_ethtool_ops = {
    .get_drvinfo = my_get_drvinfo,
    .get_link   = my_get_link,
    .get_coalesce = my_get_coalesce,
    .set_coalesce = my_set_coalesce,
};


```

★ 10. Link Management: struct phy_device

If NIC uses external PHY:

- Driver interacts via PHY layer
- Configured via MDIO interface

Functions:

phy_connect()
phy_start()
phy_stop()
phy_disconnect()

PHY provides:

- link up/down
- speed
- duplex
- auto-negotiation

★ 11. Platform / PCI Probe

Ethernet driver starts from probe function:

```
static int my_probe(struct platform_device *pdev)
```

or

```
static int my_probe(struct pci_dev *pdev, const struct pci_device_id *id)
```

Probe function does:

31. Map hardware registers with ioremap()
32. Allocate net_device
33. Set netdev_ops
34. Setup NAPI
35. Allocate and map DMA rings
36. Request IRQ
37. Call register_netdev(ndev);

★ 12. Module Init / Exit

Same as char driver:

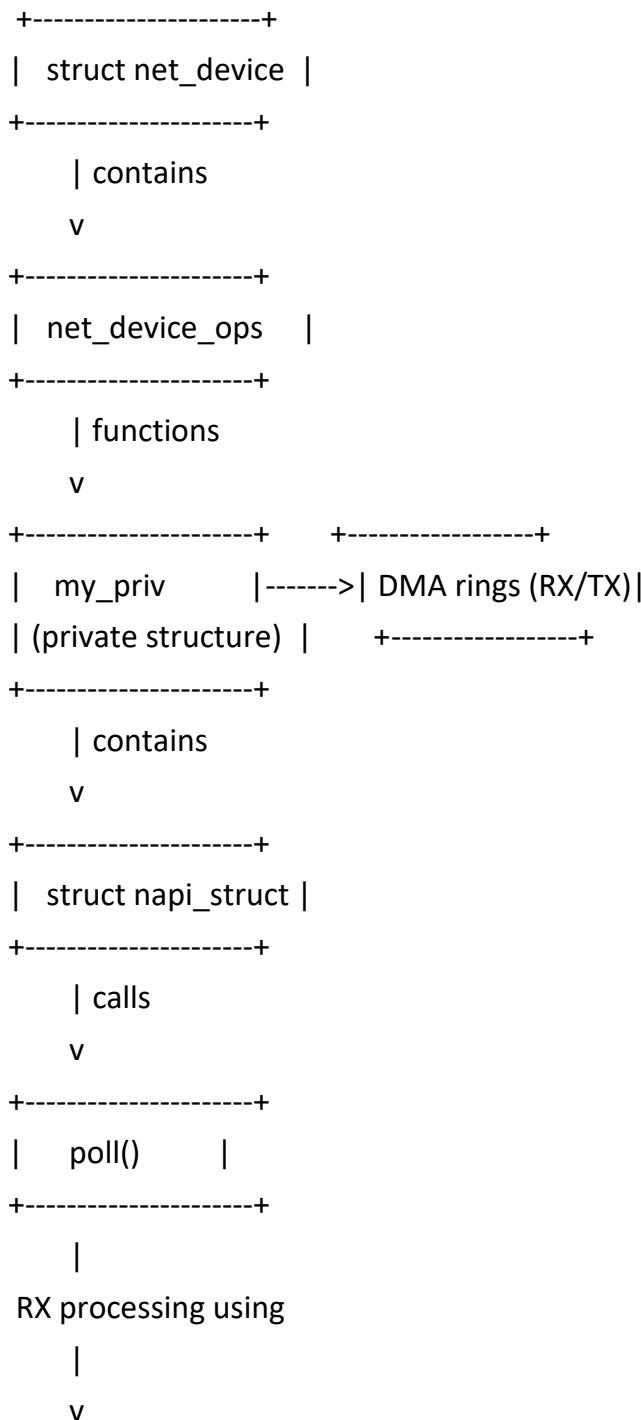
```
module_init(my_init);  
module_exit(my_exit);
```

Init registers platform/PCI driver.

Exit unregisters NIC.

★ Final Internal Architecture Diagram

Here is how all structures connect:



```
+-----+
| struct sk_buff |
+-----+
|
v
+-----+
| netif_receive_skb() |
+-----+
```

★ If you want next, I can give you:

- ✓ A fully annotated **simple Ethernet driver skeleton**
- ✓ Implementation of all the functions (open, stop, xmit, ISR, poll)
- ✓ A diagram of TX and RX flow (hardware-level)
- ✓ How these structures look in a real driver like r8169 or csw

Just tell me:

Do you want a minimal working Ethernet driver skeleton next?

Probe function

Wednesday, November 26, 2025 11:29 AM

Below is **the clearest, deepest, most theoretical explanation** of what this **probe function** does, **step-by-step**, explaining:

- What it's trying to achieve
- What information it is extracting
- Where it is extracted from
- Why each step is required
- What happens internally in the kernel

This explanation assumes **BeagleBone Black, TI AM335x, CPMAC (TI Ethernet MAC)** style driver.

★ WHAT IS THE PURPOSE OF THIS PROBE FUNCTION?

The **probe()** function is the *constructor* for the Ethernet device.

When the kernel detects a matching platform device (from Device Tree or board file), it calls:

```
cpmac_probe(struct platform_device *pdev)
```

This probe function must:

1. **Gather hardware information** from platform data / DT
2. **Locate and attach to a PHY**
3. **Allocate a net_device**
4. **Allocate + initialize private driver structure**
5. **Retrieve hardware register regions**
6. **Retrieve IRQ number**
7. **Connect the MAC <-> PHY over MDIO**
8. **Register the interface with networking stack**

Only after probe succeeds does the interface appear as:

```
eth0
```

★ STEP-BY-STEP LOGICAL EXPLANATION

1 Extract Platform Data

```
pdata = dev_get_platdata(&pdev->dev);
```

What is this?

Platform data contains **board-specific hardware information** such as:

- MAC address
- PHY mask (which PHY addresses exist on MDIO bus)
- Bus mode (MII/RMII)
- Any special hardware quirks

Where does it come from?

From:

- **Device Tree OR**
- **Board file** (old kernels)

The platform framework attaches this data to pdev->dev during boot.

Why needed?

The driver cannot know which PHY, MAC address, bus mode etc. without it.

2 Determine Which PHY to Use

The driver now must find the **correct PHY** attached on the MDIO bus.

Case 1: External or Dumb switch

```
if (external_switch || dumb_switch)
```

- The MAC is not directly connected to a PHY
- Instead connected to a fixed function switch
 - So a fake MDIO bus "fixed-0" is used
 - PHY ID is equal to the platform device ID

Case 2: Normal PHY Auto-Scan

```
for (phy_id = 0; phy_id < PHY_MAX_ADDR; phy_id++) {  
    if (!(pdata->phy_mask & (1 << phy_id)))  
        continue;
```

phy_mask tells which PHY addresses are valid (from platform data).

```
mdiobus_get_phy(cpmac_mii, phy_id)
```

Checks if a PHY exists at that address on the MDIO bus.

If a PHY exists:

```
strncpy(mdio_bus_id, cpmac_mii->id, ...)  
break;
```

The driver now knows:

- MDIO bus name (example: "0", "1")
- PHY ID (0-31)

If no PHY found:

```
phy_id == PHY_MAX_ADDR
```

- No device responded
- Fall back to "fixed-0" bus
- PHY ID = pdev->id

3 Allocate Ethernet Device (net_device)

```
dev = alloc_etherdev_mq(sizeof(*priv), CPMAC_QUEUES);
```

What this does theoretically:

- Allocates memory for:
- struct net_device
- Driver private data (struct cpmac_priv)
- Creates queue structures
- Prepares internal networking stack structures

4 Associate net_device with platform device

```
SET_NETDEV_DEV(dev, &pdev->dev);
platform_set_drvdata(pdev, dev);
```

Purpose:

- Links the OS device model with the network device
- Allows later retrieval via platform_get_drvdata()

5 Get Private Data

```
priv = netdev_priv(dev);
```

This gives access to driver internal state storage.

6 Get Register Memory Region (MMIO)

```
mem = platform_get_resource_byname(pdev, IORESOURCE_MEM, "regs");
```

What this does:

Gets memory resource from Device Tree / board file describing:

- Register base address
- Register length

Example (from DT):

```
reg = <0x4a100000 0x2000>;
```

This tells the driver:

"Ethernet controller registers start at this physical address."

7 Get IRQ Number

```
dev->irq = platform_get_irq_byname(pdev, "irq");
```

Extracts the IRQ number from DT:

```
interrupts = <72>;
```

```
interrupt-names = "irq";
```

8 Assign Operation Callbacks

```
dev->netdev_ops = &cpmac_netdev_ops;  
dev->ethtool_ops = &cpmac_ethtool_ops;
```

These structures tell Linux how to operate the device:

- open()
- stop()
- start_xmit()
- set_mac_address()
- ethtool operations for link stats, registers, etc.

9 Add NAPI Handler

```
netif_napi_add(dev, &priv->napi, cpmac_poll, 64);
```

Purpose:

Enables NAPI soft-IRQ polling for RX packets instead of per-packet interrupts.

10 Initialize Driver Private Structure

```
spin_lock_init(&priv->lock);  
spin_lock_init(&priv->rx_lock);  
priv->ring_size = 64;  
priv->msg_enable = netif_msg_init(debug_level, 0xff);
```

Setup internal locks, ring size, debugging flags.

Copy MAC address from platform data:

```
memcpy(dev->dev_addr, pdata->dev_addr, ...);
```

1 1 Build PHY Name

```
snprintf(priv->phy_name, MII_BUS_ID_SIZE, PHY_ID_FMT,  
        mdio_bus_id, phy_id);
```

Creates something like:

"0:01" (bus 0, phy address 1)

1 2 Connect to PHY Device

```
phydev = phy_connect(dev, priv->phy_name, cpmac_adjust_link,  
                     PHY_INTERFACE_MODE_MII);
```

What happens:

- Attaches MAC to PHY framework
- Configures link callbacks
- Sets interface mode (MII/GMI/RMII)
- PHY state machine begins auto-negotiation

If PHY connect fails → driver exits.

1 3 Register the Network Device

`register_netdev(dev);`

What this does theoretically:

- Exposes interface to user space as eth0
- Creates sysfs entries
- Connects to networking core
- Allows ifconfig, ip link, ethtool, etc.

1 4 Print Debug Info

`dev_info(... "regs: %p, irq: %d, phy: %s, mac: %pM\n" ...)`

Shows important hardware data.

★ FINAL SUMMARY — WHAT DOES PROBE ACHIEVE?

The probe function:

- ✓ Extracts hardware configuration
(from platform data + device tree)
- ✓ Finds which PHY is connected
(auto-scan or fixed bus)
- ✓ Allocates the network interface
(net_device + private data)
- ✓ Gets register base and IRQ
(from DT resources)
- ✓ Initializes NAPI, locks, queues
(driver internal structures)
- ✓ Connects MAC <-> PHY
(for link detection and speed/duplex negotiation)
- ✓ Registers the interface
(makes eth0 available to the OS)

★ WHERE DOES EACH INFORMATION COME FROM?

Information	Source
Platform data (plat_cpmac_data)	Device Tree or board file
PHY mask	platform data
MDIO bus	cpmac_mii MDIO controller
PHY IDs	MDIO scan
Memory region	Device Tree (reg = <...>)
IRQ number	Device Tree (interrupts = <...>)
MAC address	platform data
PHY mode (MII/RMII)	platform data / DT
Debug level	module parameter

If you want, I can also explain:

- ✓ struct plat_cpmac_data fields in deep detail
- ✓ Entire life cycle of the net_device
- ✓ How PHY state machine works
- ✓ How MDIO bus scanning works
- ✓ How packet TX/RX rings are set up

Just tell me **which one** you want next.

TCP dump working

Friday, November 28, 2025 2:48 PM

Here is the **full, deep, step-by-step explanation of how tcpdump works internally, from user space → kernel → NIC hardware → BPF filtering → packet copy → tcpdump output.**

I will explain it like a beginner but with **kernel-level detail**.

💡 Overview — What happens when you run tcpdump

Example:

```
tcpdump -i eth0
```

This triggers a sequence of operations:

1. **tcpdump opens a raw capture socket**
2. **It attaches a BPF filter** to the socket
3. Kernel receives packets from NIC
4. NIC/DMA → SKB created → network stack
5. **Before the packet is delivered to protocol stack, BPF runs**
6. If packet matches filter → **copy goes to tcpdump**
7. tcpdump prints it in human-readable format

We will now break down each step.

❖ STEP 1 — tcpdump opens a PACKET socket

tcpdump uses:

```
socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

This is a **raw packet socket**, sitting at **Layer 2**, capturing:

- Ethernet headers
- IP
- TCP/UDP
- Payload

This is done through **PF_PACKET** interface in the Linux kernel.

❖ STEP 2 — Bind to an interface

```
bind(sock, "eth0")
```

This tells Linux:

Deliver copies of packets arriving on **eth0** to this socket.

❖ STEP 3 — tcpdump sets a BPF filter

Example:

```
tcpdump tcp port 80
```

tcpdump compiles the filter to **BPF (Berkeley Packet Filter) bytecode**.

Kernel enforces this program through **SO_ATTACH_FILTER**.

BPF filters run in kernel space on every packet.

Why?

- ✓ Avoid copying ALL packets to user space
- ✓ Process filtering in kernel (much faster)
- ✓ Drop packets not needed by tcpdump

❖ STEP 4 — NIC receives a packet

8. Packet arrives at NIC hardware
9. NIC writes packet into memory (via **DMA**)
10. NIC triggers an interrupt
 - Or NAPI polls driver
11. Driver allocates SKB
12. Packet is stored in skb->data

Now packet is ready for the network stack.

❖ STEP 5 — KERNEL MAKES A COPY FOR PACKET SOCKETS

Before passing skb to IPv4 layer, Linux checks:

packet_rcv() → For raw/packet sockets

packet_rcv_spkt() → For SOCK_DGRAM sockets

Every PF_PACKET socket receives a **copy of the skb**.

BUT...

Kernel first runs the **BPF filter**:

```
if (run_filter(skb, filter_program) != 0)
    deliver_copy_to_tcpdump();
```

```
else
    drop_copy();
```

This avoids wasting work on packets we don't care about.

❖ STEP 6 — SKB copy is queued to tcpdump socket

The kernel creates a new skb:

`skb_clone()` or `skb_copy()`

Depending on mode:

Mode	System call	Behavior
Immediate capture	<code>read()</code>	Kernel pushes packets one by one
Memory-mapped capture	<code>PACKET_MMAP</code> / ring buffer	No copy (zero-copy)

tcpdump supports **zero-copy** using `PACKET_MMAP`:

- ✓ Much faster
- ✓ NIC writes into ring buffer
- ✓ tcpdump reads directly
- ✓ No extra skb copy overhead

❖ STEP 7 — tcpdump reads packets via `read()`

tcpdump calls:

`read(sock, buf, ...)`

Kernel returns:

- Ethernet header
- IP header
- TCP/UDP header
- Payload
- Timestamps

tcpdump then uses **libpcap** to decode protocol layers:

- ✓ Ethernet
- ✓ ARP
- ✓ IPv4 / IPv6
- ✓ TCP / UDP

✓ DNS / HTTP / TLS etc.

libpcap formats it for printing.

❖ STEP 8 — tcpdump prints packet

Example printed output:

12:00:01.123456 IP 192.168.1.10.34567 > 172.217.22.14.443: Flags [S], seq 12345, win 64240

tcpdump extracted:

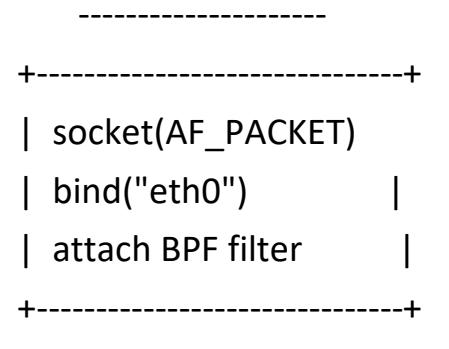
- Timestamp
- Protocol
- Source/destination addresses
- Ports
- Flags
- Sequence numbers

All from the packet returned by kernel.

=====

💧 FULL FLOWCHART (VERY IMPORTANT)

User Space (tcpdump)



|

∨

Linux Kernel (packet socket layer)

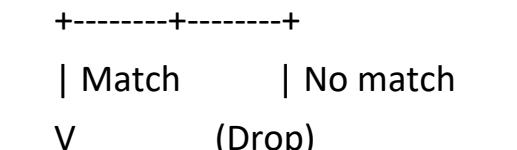
Receive incoming skb

|

∨

Run BPF filter on skb

|



Copy skb for tcpdump

|

V

Queue to socket buffer

|

V

read() / mmap ring buffer

|

V

tcpdump formats + prints output

=====

🧠 Why tcpdump is FAST?

Because:

- ✓ BPF runs *in kernel* → filters early
- ✓ Uses PF_PACKET sockets → direct Ethernet-level access
- ✓ PACKET_MMAP → zero-copy capture
- ✓ No expensive protocol processing
- ✓ No TCP/IP stack overhead

=====

🧠 What actually captures packets?

Not tcpdump.

Linux network stack captures packets.

tcpdump only:

- requests packet copies
- filters them through BPF
- prints them

=====

📣 Summary (One Sentence)

tcpdump works by creating a PF_PACKET socket, attaching a kernel BPF filter, receiving copies of packets pushed by the kernel before protocol processing, and printing parsed output using libpcap.

If you want, I can also explain:

- ◊ How BPF assembly works
- ◊ How PACKET_MMAP ring buffer works
- ◊ How tcpdump captures TX packets
- ◊ How NAPI affects packet capture
- ◊ How to write your own tcpdump-like tool

Just tell me:

“Explain BPF filter” or “Explain NAPI capture path” or “Explain PACKET_MMAP”

Below is **the most complete, beginner-friendly but deep explanation of tcpdump logs, flags, fields, timestamps, headers, and packet meaning** you will ever see.

You will understand **EVERY SINGLE CHARACTER** printed by tcpdump.

★ WHAT YOU WILL LEARN

- ✓ What every tcpdump line means
- ✓ All TCP flags
- ✓ seq, ack, win, length
- ✓ SACK, MSS, WS, Options
- ✓ Retransmissions, Out-of-order, Dup-ACK
- ✓ Timestamps, relative timestamps
- ✓ DNS logs, UDP logs
- ✓ ICMP logs
- ✓ Examples decoded

1. TCPDUMP LINE FORMAT (MOST COMMON)

A tcpdump line:

14:56:44.850830 IP 104.21.15.154.https > harshpc.38820: Flags [P.], seq 3033208325:3033208362, ack 1486715424, win 709, length 37

Breakdown:

Field	Meaning
14:56:44.850830	Timestamp (HH:MM:SS.microseconds)
IP	Protocol family (could also be ARP, ICMP, etc)
104.21.15.154.https	Source IP + source port
>	Direction of flow
harshpc.38820	Destination IP + port

Flags [P.]	TCP flags
seq 3033208325:3033208362	Sequence number range
ack 1486715424	Acknowledgement number
win 709	Receiver window size
length 37	Data payload

That's the complete layout.

2. TCP FLAGS — THE MOST IMPORTANT PART

tcpdump shows flags inside brackets:

Flags [S]

Flags [S.]

Flags [P.]

Flags [.]

Flags [F.]

Flags [R]

Here is the **full meaning**:

Flag	Letter	Meaning
SYN	S	Establish a connection
ACK	.	Normal acknowledgement
FIN	F	Gracefully close connection
RST	R	Reset connection (abort)
PSH	P	Push data immediately
URG	U	Urgent pointer valid
ECE	E	Explicit Congestion Notification Echo
CWR	C	Congestion Window Reduced
NS	N	ECN Nonce

3. MOST COMMON FLAG COMBINATIONS

- SYN — Starting a new connection

Flags [S]

- SYN + ACK — Server responds to SYN

Flags [S.]

- ACK only — Normal acknowledgement

Flags [.]

- PSH + ACK — Data being sent immediately

Flags [P.]

Used for:

- small TCP messages
- HTTP headers
- TLS handshake
 - FIN + ACK — Connection terminating

Flags [F.]

● RST — Connection forcibly reset

Flags [R]

4. SEQUENCE NUMBERS (seq)

tcpdump prints sequence numbers in two ways:

✓ Initial SYN

seq 0

This is normalized.

✓ Normal data packet

seq 3033208325:3033208362

This means:

- First byte = **3033208325**
- Last byte = **3033208362 - 1**
- Payload = **37 bytes**

tcpdump prints the range of bytes sent.

5. ACKNOWLEDGEMENT NUMBER (ack)

ack 1486715424

Meaning:

"I have received all bytes up to **1486715424-1**"

TCP is *byte based*, not packet based.

6. WINDOW SIZE (win)

win 709

Meaning:

Receiver can currently accept **709 more bytes**.

Used for TCP flow control.

7. TCP LENGTH (payload)

length 37

Exactly the amount of application data (not TCP header).

8. COMMON TAGS tcpdump ADDS

 “[tcp sum ok]” or “[bad udp cksum]”

Checksum verification.

 “[retransmission]”

Kernel detected a resent packet.

 “[dup ack]”

Receiver is asking to resend a packet.

 “[out-of-order]”

Packet received earlier/later than expected.

 “[ECN]”

Congestion control flags.

9. SPECIAL TCP PACKETS

 Retransmission

[Retransmission]

seq 1000:2000

Means packet was resent.

 Duplicate ACK

Flags [.], ack 2000 win ...

[dup ack]

Meaning:

Receiver got a gap and is asking for retransmission repeatedly.

 Out of order

[out-of-order]

Example: seq 2000 arrives before seq 1000.

Fast Retransmit

After 3 duplicate ACKs, sender retransmits.

10. ICMP PACKET LOGS

IP 8.8.8.8 > 192.168.1.2: ICMP echo request

- Echo request = ping
- Echo reply = pong

11. UDP PACKET LOGS

IP 192.168.1.10.55324 > 8.8.8.8.domain: 12345+ A? google.com. (28)

This is a DNS query:

- Your port → DNS port 53
- ID 12345
- Query type A (IPv4)

12. DNS RESPONSE

12345 1/0/0 A google.com 142.250.78.206

Format:

- 1 Answer
- 0 Authority
- 0 Additional

13. ARP PACKETS

ARP, Request who-has 192.168.1.1 tell 192.168.1.10

Meaning:

"Who owns 192.168.1.1? I am 192.168.1.10"

ARP, Reply 192.168.1.1 is-at aa:bb:cc:dd:ee:ff

14. HTTP / HTTPS (TLS) LINES

HTTPS is encrypted.

So you only see packet lengths, not content.

Typical:

Flags [P.], length 342

HTTP (port 80) can show ASCII data with -A option:

GET /index.html HTTP/1.1

15. COMPLETE REAL-WORLD FLOW

TCP Three-Way Handshake

S

S.

.

Data Transfer

P.

.

P.

.

Connection Close

F.

.

F.

.



BONUS: SECRET TRICKS FOR READING TCPDUMP LIKE A PRO

- ✓ ACK number = next byte expected
- ✓ seq range shows how much data was sent
- ✓ length = payload only
- ✓ window controls speed
- ✓ lots of duplicate ACKs = packet loss
- ✓ RST = abrupt close
- ✓ S and S. = new connection



IF YOU WANT:

I can generate:

- ✓ A cheat-sheet PDF
- ✓ Colored tcpdump decoder script
- ✓ C program that parses tcpdump logs
- ✓ Kernel-level explanation of TCP more deeply
- ✓ A Wireshark-like output in terminal

Just tell me!

