A **Linux Platform Driver** is a type of device driver used in the Linux kernel to manage platform-specific hardware devices that do not fall under standard bus interfaces like PCI or USB. These devices are typically integrated into the system-on-chip (SoC) and do not have dynamically discoverable buses. The **Linux Platform Driver Model** is designed for non-discoverable devices and follows a simple mechanism:

1. **Platform Device**: Represents a hardware device that does not use a discoverable bus.
2. **Platform Driver**: A software module that provides functionality for the corresponding platform device.
3. **Device Tree / Board Files**: Used to describe platform devices in embedded systems.

**Key Components of the Model**

* **struct platform\_device**: Represents a device in the system.
* **struct platform\_driver**: Defines the driver that manages the device.
* **platform\_register\_driver()**: Registers the platform driver with the kernel.
* **probe() function**: Invoked when a matching platform device is found.
* **remove() function**: Handles device removal.

**Other Driver Models in Linux**

Apart from the Platform Driver Model, Linux supports several other driver models:

**1. Bus-Based Driver Models**

* **PCI Driver Model**: For devices connected via the PCI bus.
* **USB Driver Model**: For USB devices.
* **I2C Driver Model**: For devices on the I2C bus.
* **SPI Driver Model**: For SPI-connected peripherals.

**2. Character and Block Device Models**

* **Character Drivers**: Used for devices like serial ports (/dev/ttyS0).
* **Block Drivers**: Used for storage devices like HDDs (/dev/sda).

**3. Virtual Driver Models**

* **Virtio Driver Model**: For virtualized environments (used in KVM, QEMU).
* **VFS (Virtual File System) Model**: Used for filesystem abstraction.

**4. Network Driver Model**

* **NET Device Model**: For network interfaces like Ethernet (struct net\_device).
* **Wi-Fi Driver Model**: Used for wireless devices.

In Qualcomm, both **PCIe platform drivers** and **PCIe bus drivers** are used because they serve different purposes within the Linux kernel’s driver model. Here's why:

**1. PCIe Bus Driver (Generic)**

* The **PCIe bus driver** is a standard Linux kernel component that handles PCIe **device enumeration, configuration, and communication**.
* It follows the **PCIe subsystem model**, where:
  + The **PCIe root complex (RC)** acts as a bus controller.
  + PCIe devices (endpoints) are detected dynamically.
* The PCIe **bus driver is responsible for managing devices dynamically discovered** on the PCIe bus.
* It uses the standard PCI driver API (struct pci\_driver).

✔ **Why?** Because PCIe is a discoverable bus, and Qualcomm SoCs may connect to different PCIe-based peripherals dynamically, requiring the kernel's PCI subsystem to handle enumeration.

**2. PCIe Platform Driver (Custom)**

* A **platform driver** in Qualcomm’s PCIe implementation is used for **initializing the PCIe controller (root complex) embedded in the SoC**.
* The root complex is a **non-discoverable hardware block**, and it is **not detected automatically by the PCIe bus driver**.
* The platform driver:
  + Initializes the PCIe controller (power, clocks, reset).
  + Configures PHY and link training.
  + Registers the controller with the Linux PCI subsystem so the bus driver can enumerate devices.

✔ **Why?** Because the **PCIe controller in Qualcomm SoCs is not on a discoverable bus**, it requires a **platform driver to bring it up before the PCI subsystem can function**.

**How They Work Together**

1. **Platform Driver (PCIe Controller)**
   * Brings up the PCIe root complex (RC).
   * Configures PCIe clocks, resets, and power.
   * Registers the RC with the PCI subsystem.
2. **Bus Driver (PCIe Enumeration)**
   * Once the RC is active, the **PCIe bus driver detects and initializes endpoint devices** (Wi-Fi, storage, etc.).
   * Loads the appropriate PCIe device drivers.

**Conclusion**

* **Platform driver** → Initializes **PCIe Root Complex** (SoC-specific).
* **Bus driver** → Handles **PCIe device enumeration and communication**.

**1. Are Bus Drivers for Self-Discoverable Devices?**

Yes, **bus drivers are primarily used for self-discoverable devices**. In Linux, a **bus driver** manages a hardware bus where devices are dynamically detected and enumerated. The kernel's bus framework (like PCI, USB, I2C) automatically discovers and registers devices connected to the bus.

**2. Difference Between Driver Models and When to Use Them**

|  |  |  |  |
| --- | --- | --- | --- |
| **Driver Model** | **Discovery** | **Use Case** | **Example** |
| **Bus Driver** | Self-discoverable | Handles devices on dynamic buses. Best for devices that can **self-identify** and are connected via a standard bus. If the device is attached to a **discoverable bus**, the Linux kernel already has a subsystem to manage it. | PCIe, USB, I2C, SPI |
| **Platform Driver** | Non-discoverable | Manages fixed hardware components. If the device is part of the **fixed hardware** and needs manual registration in the device tree. | SoC components, PCIe root complex, GPIO controllers, UART, power controller |
| **Character Driver** | Non-discoverable | Provides user-space interaction for character devices. Used when a device provides **byte-stream** data access. | Serial ports (/dev/ttyS0), sensors, touchscreen |
| **Block Driver** | Non-discoverable | Manages block storage devices. If the device requires **block-based read/write operations.** | HDDs, SSDs (/dev/sda), eMMC, SD card |
| **Network Driver** | Either | Handles network interfaces | Ethernet, Wi-Fi (struct net\_device) |
| **Virtual Driver** | Depends | Manages virtualized hardware | Virtio, Hypervisor drivers, Hyper-V synthetic drivers, Docker, VM network drivers |

**Qualcomm PCIe Example**

* **PCIe Controller (Root Complex) → Platform Driver**
  + The **root complex is fixed** and does not self-enumerate.
  + It must be initialized manually in the **Device Tree**.
* **PCIe Endpoints (Wi-Fi, SSD) → PCIe Bus Driver**
  + The PCIe **bus driver discovers** and manages endpoint devices dynamically.

**5. Summary**

* **If the device is on a self-discoverable bus** → Use **Bus Driver**.
* **If the device is fixed in the hardware (SoC)** → Use **Platform Driver**.
* **If the device interacts with user space (byte-stream)** → Use **Character Driver**.
* **If the device stores data in blocks** → Use **Block Driver**.
* **If the device is a network interface** → Use **Network Driver**.
* **If the device is in a virtualized system** → Use **Virtual Driver**.

Not all driver models require a **Device Tree (DT)**, but many do—especially in **embedded systems and ARM-based SoCs** like Qualcomm's. Let's break down **which models use Device Tree, how they use it, and why some don't.**

**1. Which Driver Models Use Device Tree?**

|  |  |  |
| --- | --- | --- |
| **Driver Model** | **Uses Device Tree?** | **Why?** |
| **Platform Driver** | ✅ Yes | Needed to describe fixed hardware in SoCs (e.g., PCIe root complex, GPIOs, clocks). |
| **Bus Driver** (PCIe, I2C, SPI) | ✅ Yes (sometimes) | Used when the bus itself needs configuration (e.g., PCIe controllers in SoCs). |
| **Character Driver** | ❌ No (mostly) | Devices like serial ports often don't need DT, but some embedded devices do (e.g., sensors over I2C). |
| **Block Driver** | ❌ No | Storage devices (eMMC, SD) are usually detected via standard mechanisms. |
| **Network Driver** | ✅ Yes (for SoC NICs) | SoC Ethernet controllers need DT, but PCI/USB NICs don’t. |
| **Virtual Driver** | ❌ No | Virtual devices (e.g., Virtio, Hyper-V) don’t need DT as they are software-based. |

**2. How Do Drivers Use Device Tree?**

**A) Platform Drivers (Most Common in DT)**

Platform drivers rely on the **Device Tree (DT) to get hardware configuration** since the hardware is not discoverable.

**Example: PCIe Root Complex in Qualcomm SoC**

In **Device Tree (.dts file)**:

pcie0: pcie@1c00000 {

compatible = "qcom,pcie"; // Matches driver

reg = <0x1c00000 0x1000>; // PCIe memory region

clocks = <&gcc PCIe\_clk>; // Clock dependency

resets = <&gcc PCIe\_rst>; // Reset line

status = "okay";

};

In **Platform Driver (pcie-qcom.c)**:

static const struct of\_device\_id qcom\_pcie\_dt\_ids[] = {

{ .compatible = "qcom,pcie", },

{}

};

MODULE\_DEVICE\_TABLE(of, qcom\_pcie\_dt\_ids);

static int qcom\_pcie\_probe(struct platform\_device \*pdev) {

struct resource \*res;

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

if (!res)

return -EINVAL;

// Map PCIe memory

void \_\_iomem \*pcie\_base = devm\_ioremap\_resource(&pdev->dev, res);

// Initialize clocks, reset, etc.

return 0;

}

static struct platform\_driver qcom\_pcie\_driver = {

.probe = qcom\_pcie\_probe,

.driver = {

.name = "qcom\_pcie",

.of\_match\_table = qcom\_pcie\_dt\_ids,

},

};

module\_platform\_driver(qcom\_pcie\_driver);

**How DT is used?**

* The driver **matches the compatible property** (qcom,pcie).
* Reads **register addresses, clocks, and resets** from DT.
* Initializes the **PCIe root complex**.

**B) Bus Drivers (I2C, SPI, Sometimes PCIe)**

* If a bus controller itself is **not discoverable**, it needs a DT entry.
* But **devices on the bus (I2C sensors, PCIe SSDs)** are **self-enumerating** and do **not need DT**.

✔ **Example: I2C Controller (Needs DT)**

i2c0: i2c@0 {

compatible = "qcom,i2c";

reg = <0x07850000 0x1000>;

clock-frequency = <100000>;

status = "okay";

};

✔ **Example: I2C Sensor (May Need DT)**

temperature\_sensor@48 {

compatible = "bosch,bmp280";

reg = <0x48>;

};

**How DT is used?**

* **The I2C bus itself** is described in DT.
* The I2C **bus driver reads DT to initialize the controller**.
* Sensors can either be in DT **or probed dynamically**.

**C) Network Drivers (Sometimes Needs DT)**

* **PCI/USB network cards don’t need DT** (they are discoverable).
* **SoC Ethernet controllers require DT** to configure MAC addresses, PHY settings, etc.

**Example: Ethernet Controller in DT**

eth0: ethernet@0x1a00000 {

compatible = "qcom,gmac";

reg = <0x1a00000 0x1000>;

phy-handle = <&phy0>;

};

✔ **How DT is used?**

* The **Ethernet driver fetches DT properties** like registers and PHY settings.

**D) Character and Block Drivers (Usually No DT)**

* **Character devices (e.g., serial ports) and block devices (e.g., eMMC, SD cards) are usually managed by standard subsystems**.
* **Exception**: If a character device is on an SoC peripheral (e.g., UART), **DT may define the port**.

✔ **Example: UART in DT**

uart0: serial@07870000 {

compatible = "qcom,serial";

reg = <0x07870000 0x1000>;

status = "okay";

};

✔ **How DT is used?**

* The **serial driver reads DT to find hardware addresses** and initializes UART.

**3. Why Some Drivers Don’t Use DT?**

1. **Self-Discoverable Buses (PCI, USB, etc.)**
   * **Don’t need DT** because hardware is detected at runtime.
   * Example: A PCIe Wi-Fi card doesn't need DT because the PCIe subsystem detects it.
2. **Standard Devices (Storage, Generic Peripherals)**
   * Hard drives, SSDs, and USB devices **don’t need DT** since they use well-defined protocols.
3. **Virtual Devices (Virtio, Hyper-V, Docker, etc.)**
   * Virtual hardware is defined in software and doesn’t need a hardware description.

**5. When to Use Device Tree?**

✔ **Use DT if:**

* The hardware **is part of an SoC (fixed, non-discoverable)**.
* The device **needs special initialization (clocks, regulators, GPIOs)**.
* It’s an **I2C, SPI, or network interface that isn't discoverable**.

❌ **Don't use DT if:**

* The device **is on a discoverable bus (PCI, USB, etc.)**.
* The device **is purely virtual or software-defined**.
* It’s a standard **block or character device** managed by subsystems.

Yes, **platform drivers use the probe function** to start their operation when a matching device is found in the **Device Tree (DT) or ACPI tables**. However, other driver models have different initialization mechanisms. Below is a comparison of how different driver models start their operation:

**Driver Models and Their Initialization Functions**

|  |  |  |
| --- | --- | --- |
| **Driver Model** | **Initialization Function** | **When It Gets Called?** |
| **Platform Driver** | probe(struct platform\_device \*pdev) | When a device matching the **Device Tree (DT)** or ACPI table is found. |
| **Bus Driver** (PCI, USB, I2C, SPI) | probe(struct pci\_device \*pdev), probe(struct usb\_interface \*intf), probe(struct i2c\_client \*client) | When the device is detected on the respective **bus**. |
| **Character Driver** | open(struct inode \*inode, struct file \*filp) | When a user-space application opens the device file (/dev/mydevice). |
| **Block Driver** | request\_queue\_fn(struct request\_queue \*q, struct request \*req) | When the OS issues a read/write request to the block device. |
| **Network Driver** | ndo\_open(struct net\_device \*dev) | When the network interface is brought up (ifconfig eth0 up). |
| **Virtual Driver** | probe(struct virtio\_device \*vdev) | When the virtual device is initialized by the hypervisor. |

**Detailed Explanation of Each Case**

**1. Platform Driver (Uses probe)**

✔ **When?**

* Called when a **matching entry is found in the DT (compatible)**.
* Registers the driver to manage a **non-discoverable device** (e.g., SoC peripherals, PCIe root complex).

✔ **Example: Platform Driver (my\_pci\_controller.c)**

static int my\_pci\_probe(struct platform\_device \*pdev) {

printk("PCIe Controller Initialized\n");

return 0;

}

static const struct of\_device\_id my\_pci\_of\_match[] = {

{ .compatible = "qcom,my\_pci\_controller" },

{ }

};

static struct platform\_driver my\_pci\_driver = {

.probe = my\_pci\_probe,

.driver = {

.name = "my\_pci",

.of\_match\_table = my\_pci\_of\_match,

},

};

module\_platform\_driver(my\_pci\_driver);

✔ **Key Points:**

* probe() is called **when the DT entry compatible = "qcom,my\_pci\_controller" is found**.
* Platform drivers don’t have dynamic discovery—**they rely on DT or ACPI**.

**2. Bus Driver (PCI, USB, I2C, SPI) (Uses probe)**

✔ **When?**

* Called when the bus subsystem detects a device.
* **Unlike platform drivers, bus drivers work with dynamically discovered devices.**

✔ **Example: PCIe Bus Driver**

static int my\_pci\_probe(struct pci\_dev \*pdev, const struct pci\_device\_id \*id) {

printk("PCIe Device Found: %x\n", pdev->device);

return 0;

}

static const struct pci\_device\_id my\_pci\_table[] = {

{ PCI\_DEVICE(0x8086, 0x1234) }, // Example Vendor/Device ID

{ }

};

static struct pci\_driver my\_pci\_driver = {

.name = "my\_pci\_device",

.id\_table = my\_pci\_table,

.probe = my\_pci\_probe,

};

module\_pci\_driver(my\_pci\_driver);

✔ **Key Points:**

* probe() is called **when a PCIe device with Vendor ID 0x8086 and Device ID 0x1234 is detected**.
* Unlike platform drivers, **the device is discovered at runtime**.

**3. Character Driver (Uses open)**

✔ **When?**

* Called **when a user-space program opens the device file** (/dev/mychardev).

✔ **Example: Character Driver**

static int mychar\_open(struct inode \*inode, struct file \*filp) {

printk("Character Device Opened\n");

return 0;

}

static struct file\_operations my\_fops = {

.owner = THIS\_MODULE,

.open = mychar\_open,

};

static int \_\_init mychar\_init(void) {

register\_chrdev(240, "mychardev", &my\_fops);

return 0;

}

module\_init(mychar\_init);

✔ **Key Points:**

* **No probe() function** because there is **no hardware discovery**.
* Instead, it registers a device file (/dev/mychardev), and **open() is called when a user-space process accesses it**.

**4. Block Driver (Uses request\_queue\_fn)**

✔ **When?**

* Called **when the OS needs to read/write a block of data**.

✔ **Example: Block Driver**

static void my\_block\_request(struct request\_queue \*q) {

struct request \*req;

while ((req = blk\_fetch\_request(q)) != NULL) {

printk("Handling Block Request\n");

\_\_blk\_end\_request\_all(req, 0);

}

}

static int my\_block\_init(void) {

struct request\_queue \*queue = blk\_init\_queue(my\_block\_request, NULL);

return 0;

}

module\_init(my\_block\_init);

✔ **Key Points:**

* No probe() because **block devices are not discovered dynamically**.
* Instead, the function is triggered **when a read/write operation occurs**.

**5. Network Driver (Uses ndo\_open)**

✔ **When?**

* Called **when the network interface is activated (ifconfig eth0 up)**.

✔ **Example: Network Driver**

static int my\_net\_open(struct net\_device \*dev) {

printk("Network Interface Opened\n");

return 0;

}

static struct net\_device\_ops my\_netdev\_ops = {

.ndo\_open = my\_net\_open,

};

static void my\_net\_setup(struct net\_device \*dev) {

dev->netdev\_ops = &my\_netdev\_ops;

}

static int \_\_init my\_net\_init(void) {

struct net\_device \*dev = alloc\_netdev(0, "eth0", NET\_NAME\_UNKNOWN, my\_net\_setup);

register\_netdev(dev);

return 0;

}

module\_init(my\_net\_init);

✔ **Key Points:**

* No probe(), because **network interfaces are registered manually**.
* Instead, **ndo\_open() is called when the network device is activated**.

**6. Virtual Driver (Uses probe)**

✔ **When?**

* Called when the **hypervisor initializes a virtual device (e.g., Virtio, Hyper-V)**.

✔ **Example: Virtio Driver**

static int my\_virtio\_probe(struct virtio\_device \*vdev) {

printk("Virtio Device Initialized\n");

return 0;

}

static struct virtio\_driver my\_virtio\_driver = {

.probe = my\_virtio\_probe,

.driver.name = "my\_virtio",

};

module\_virtio\_driver(my\_virtio\_driver);

✔ **Key Points:**

* Virtual devices **don’t exist physically** but still use probe() to be initialized by the hypervisor.

**Summary of Initialization Functions**

|  |  |  |
| --- | --- | --- |
| **Driver Model** | **Initialization Function** | **Trigger Condition** |
| **Platform Driver** | probe(struct platform\_device \*pdev) | When a matching DT entry is found. |
| **Bus Driver** (PCI, USB, I2C, SPI) | probe(struct pci\_dev \*pdev) | When a device is detected on the bus. |
| **Character Driver** | open(struct inode \*inode, struct file \*filp) | When a user-space process opens /dev/mydevice. |
| **Block Driver** | request\_queue\_fn(struct request\_queue \*q, struct request \*req) | When the OS sends a block I/O request. |
| **Network Driver** | ndo\_open(struct net\_device \*dev) | When the network interface is brought up. |
| **Virtual Driver** | probe(struct virtio\_device \*vdev) | When the hypervisor initializes the device. |