USB Subsystem & Device Tree

Part 1: USB Subsystem

1. Introduction to USB

The Universal Serial Bus (USB) is an industry standard for short-distance data communication between a host (PC or embedded system) and peripherals (mouse, keyboard, storage, etc.).
In Linux, the USB subsystem provides support for both USB host and device roles, implemented in kernel space under `drivers/usb/

USB Protocol and Linux USB Core

USB Protocol Overview

The Universal Serial Bus (USB) defines how devices communicate with a host system. It standardizes electrical, signaling, and protocol layers to ensure compatibility between devices.

USB Protocol Layers:

1. Physical Layer: Defines cables, connectors, and signaling.
2. Protocol Layer: Handles transactions, packets, and transfers.
3. Device Framework Layer: Defines descriptors, endpoints, and configurations.

USB Data Flow Types:

- Control Transfers: Used for configuration and command exchanges.
- Bulk Transfers: Large data transfers (e.g., mass storage).
- Interrupt Transfers: Small, time-critical data (e.g., keyboards, mice).
- Isochronous Transfers: Time-bound data (e.g., audio, video streaming).

USB Device Types

USB defines classes to group similar device functionalities. Each class has a Class Code defined by USB-IF.

Device Type	Example	Class Code
Mass Storage	Pendrive	0x08
Human Interface Device (HID)	Keyboard, Mouse	0x03
Communication Device	Modems, Ethernet	0x02
Audio	Microphones, Speakers	0x01
Video	Webcams	0x0E
CDC-ACM	Virtual Serial Ports	0x02

Each device provides descriptors:

- Device Descriptor (general info)
- Configuration Descriptor
- Interface Descriptor
- Endpoint Descriptor

USB Roles in Linux

Linux supports three main USB operational roles:

Role	Description	Example
Host Mode	Acts as a master that controls devices.	PC, BeagleBone Black (host port)
Device Mode (Gadget)	Acts as a peripheral to another host.	USB Gadget framework on BBB

OTG (On-The-Go)	Can switch between Host and Device dynamically.	Mobile devices, BBB USB0 port

Host Mode

- The system enumerates devices.
- Managed by Host Controller Drivers (HCD) such as EHCI, OHCI, or XHCI.
- Directory: /drivers/usb/host/

Device Mode (Gadget)

- Implements functions like serial, mass storage, or Ethernet over USB.
- Uses UDC (USB Device Controller) drivers.
- Directory: /drivers/usb/gadget/

OTG Mode

- Combines Host and Device functionality.
- Role decided by ID pin on the micro-USB connector.
- Controlled by dual-role controllers.

Linux USB Core APIs

The USB Core is the heart of the Linux USB subsystem. It provides functions for driver registration, communication, and memory management.

Key Structures

Structure	Purpose
struct usb_driver	Represents a USB driver.
struct usb_device	Represents a connected USB device.
struct usb_interface	Represents a specific interface in a device.

struct urb	USB Request Block – used for sending/receiving data.

Common USB Core APIs

API	Purpose
usb_register() / usb_deregister()	Register/unregister USB driver with the kernel.
usb_alloc_urb() / usb_free_urb()	Allocate or free a USB Request Block.
usb_submit_urb()	Submit an URB to the USB core for processing.
usb_control_msg()	Send control messages (setup packets).
usb_bulk_msg()	Send/receive bulk data.
usb_get_dev() / usb_put_dev()	Manage USB device references.

Example: Submitting a Bulk Transfer

```
struct urb *my_urb;
void *buffer;

buffer = kmalloc(512, GFP_KERNEL);
my_urb = usb_alloc_urb(0, GFP_KERNEL);

usb_fill_bulk_urb(my_urb, usb_dev, usb_sndbulkpipe(usb_dev, endpoint),
        buffer, 512, my_completion_handler, NULL);

usb_submit_urb(my_urb, GFP_KERNEL);
```

USB Enumeration in Linux

Steps when a USB device is plugged in:

1. Host detects device connect event.
2. USB Core assigns an address.
3. Device Descriptors are read.
4. Kernel matches driver using the idVendor and idProduct fields.
5. Corresponding driver's probe() function is invoked.

2. USB Architecture and Protocol

USB uses a tiered star topology consisting of a host, hubs, and devices. It supports different speeds: Low (1.5 Mbps), Full (12 Mbps), High (480 Mbps), and SuperSpeed (5 Gbps).
USB defines four main transfer types: Control, Bulk, Interrupt, and Isochronous.

3. USB Subsystem in Linux Kernel

The Linux USB subsystem is divided into three major parts:
- USB Core: Manages enumeration and driver matching (drivers/usb/core).
- Host Controller Drivers (HCD): Handle low-level communication with hardware (e.g., EHCI, XHCI).
- Gadget Drivers: Implement USB device mode functionality (for embedded systems).

4. USB Core, Host, and Gadget Drivers

USB Core registers devices and interfaces on the USB bus. When a USB device is connected, the kernel enumerates it and matches it with a suitable driver based on the `idVendor` and `idProduct` fields.

5. Writing a Simple USB Driver

Below is a minimal USB device driver example:

```
#include <linux/module.h>
#include <linux/usb.h>

#define USB_VENDOR_ID  0x0781
#define USB_PRODUCT_ID 0x5567

static struct usb_device_id pen_table[] = {
        { USB_DEVICE(USB_VENDOR_ID, USB_PRODUCT_ID) },
        {} /* Terminating entry */
};
MODULE_DEVICE_TABLE(usb, pen_table);
```

```
static int pen_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
        printk(KERN_INFO "USB Device Plugged: Vendor ID = %04X, Product ID = %04X\n",
id->idVendor, id->idProduct);
        return 0;
}

static void pen_disconnect(struct usb_interface *interface)
{
        printk(KERN_INFO "USB Device Removed\n");
}

static struct usb_driver pen_driver = {
        .name = "pen_driver",
        .id_table = pen_table,
        .probe = pen_probe,
        .disconnect = pen_disconnect,
};

module_usb_driver(pen_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Pujit Kumar");
MODULE_DESCRIPTION("Simple USB Driver Example");
```

6. USB Gadget Framework

In embedded Linux, systems can act as USB devices (gadgets). The gadget framework allows you to create composite USB functions like mass storage, Ethernet, or serial devices using `ConfigFS`.

7. Debugging USB

Useful commands:
- `lsusb` — Lists connected USB devices.
- `dmesg | grep usb` — Shows kernel USB logs.
- `cat /sys/kernel/debug/usb/devices` — Displays USB hierarchy.
- `modprobe usbmon` — For capturing USB traffic.

8. Important Kernel Structures and APIs

Common structures:
- `struct usb_driver`
- `struct usb_device`
- `struct usb_interface`
- `struct urb` (USB Request Block)

APIs:
- `usb_alloc_urb()`, `usb_submit_urb()`, `usb_free_urb()`
- `usb_get_dev()`, `usb_put_dev()`

Part 2: Device Tree

1. Introduction and Purpose

The Device Tree (DT) describes the hardware components of a system to the Linux kernel. It is used instead of hardcoding device information into the kernel, allowing flexibility across platforms.

2. DTS, DTB, and DTC Overview

- **DTS (Device Tree Source):** Human-readable source file describing hardware.
- **DTB (Device Tree Blob):** Compiled binary version of DTS loaded by the bootloader.
- **DTC (Device Tree Compiler):** Tool that converts DTS ↔ DTB.

3. Syntax, Nodes, and Properties

Device Tree is written in a hierarchical format:
example:

```
/ {
        model = "BeagleBone Black";
        compatible = "ti,am335x-bone-black";
        memory@80000000 {
        device_type = "memory";
        reg = <0x80000000 0x10000000>;
        };
};
```

4. Common Properties

- `compatible`: Used to match the device with its driver.
- `reg`: Specifies address and size of memory-mapped I/O.
- `interrupts`: Defines IRQ numbers and trigger types.
- `gpios`: GPIO pin configurations.
- `clocks`: Defines clock sources.

5. Writing and Compiling a Device Tree

To compile:
$ dtc -I dts -O dtb -o mydevice.dtb mydevice.dts

To decompile:

```
$ dtc -I dtb -O dts -o mydevice.dts mydevice.dtb
```

6. Integrating DT with Platform Drivers

A platform driver can be linked to a DT node using the `compatible` string.

Example driver code:

```
#include <linux/module.h>
#include <linux/of.h>
#include <linux/platform_device.h>

static int my_probe(struct platform_device *pdev)
{
        printk(KERN_INFO "Device Tree driver matched\n");
        return 0;
}

static int my_remove(struct platform_device *pdev)
{
        printk(KERN_INFO "Device removed\n");
        return 0;
}

static const struct of_device_id my_dt_ids[] = {
        { .compatible = "pujit,mydevice" },
        { }
};
MODULE_DEVICE_TABLE(of, my_dt_ids);

static struct platform_driver my_driver = {
        .driver = {
        .name = "my_device_driver",
        .of_match_table = my_dt_ids,
        },
        .probe = my_probe,
        .remove = my_remove,
};

module_platform_driver(my_driver);
MODULE_LICENSE("GPL");
```

7. Device Tree Bindings and Documentation

Bindings describe how devices should be represented in DT. These are found in:
`Documentation/devicetree/bindings/`
Each binding file describes required and optional properties.

8. Debugging and Verification

- View loaded DT: `ls /proc/device-tree/`
- Dump a node: `hexdump -C /proc/device-tree/<node>/compatible`
- Common errors: invalid `reg`, missing `compatible`, wrong address mapping.