Device Tree Overlays

1. Device tree overlays bookmark_border A device tree (DT) is a data structure of named nodes and properties that describe non-discoverable hardware. Kernels, such as the Linux kernel used in Android, use DTs to support a wide range of hardware configurations used by Android-powered devices. Hardware vendors supply their own device tree source (DTS) files, which are compiled into the device tree blob (DTB) file using the device tree compiler. These files are then used by the bootloader. The DTB file contains a binary-formatted flattened device tree. A device tree overlay (DTO) enables a central DTB to be overlaid on the DT (a device tree blob for overlay (DTBO)). A bootloader using DTO can maintain the system-on-chip (SoC) DT and dynamically. 3
2. Load a DT Loading a DT in bootloader involves building, partitioning, and running.. 5
3. Figure 1. Typical implementation for loading device tree in bootloader. To create and flash the DTB: 1a. Use the DTC (dtc) to compile DTS (.dts) into a DTB (.dtb). The DTB is formatted as a flattened DT. 1b. Flash the .dtb file into a bootloader runtime-accessible location (detailed below). To partition, determine a bootloader runtime-accessible and trusted location in flash memory to put .dtb. Example locations: Figure 2. Put .dtb in a boot partition by appending to image.gz and passing as "kernel" to mkbootimg. Figure 3. Put .dtb in a unique partition (for example, dtb partition). To load the DTB and start the kernel: Load .dtb from storage into memory. Start kernel using the memory address of the loaded DT.. 7
4. Implement DTOs bookmark_border Implementing device tree overlays (DTOs) involves dividing the device tree (DT), building, partitioning, and running. After you have a working implementation, you must also maintain compatibility between the two DTs and determine a strategy for ensuring the security of each DT partition. Divide the DT Start by dividing the DT into two parts: Main DT. The SoC-only part and the default configurations, provided by SoC vendor. Overlay DT. The device-specific configurations, provided by ODM/OEM. After dividing the DTs, you must ensure compatibility between main DT and overlay DT so that merging main DT and overlay DT results in a complete DT for the device. For details on DTO format and rules, see DTO syntax. Fo. 9
5. dtc -I dts -O dtb -o main.dtb main.dtsexplain. 12
6. U-Bootwhat is this. 14
7. Partition DTs Determine a bootloader runtime-accessible and trusted location in flash memory to put .dtb and .dtbo. Example locations for the main DT: Part of boot partition, appended to the kernel (image.gz) Separate DT blobs (.dtb) in dedicated partition (dtb) Example locations for the overlay DT:. 16
8. For devices that support seamless (A/B) updates, A/B the main DT and overlay DT partitions:. 18
9. Run in bootloader Figure 5. Typical runtime implementation for DTO in bootloader. Load .dtb from storage into memory. Load .dtbo from storage into memory. Overlay .dtb with .dtbo to be a merged DT. Start kernel given the memory address of the merged DT. Maintain compatibility The main DTB (from SoC vendor) is treated as an API surface for DTBOs. After separating the DT into a SoC-common part and a device-specific part, you must keep the two parts mutually compatible in the future, including: DT definition in main DT. For example, nodes, properties, labels. Any definition change in main DT could trigger changes in overlay DT. For example, to correct a node name in main DT, define an "alias" label that maps to the original node name (to av. 20
10. AVB Trust Chainwhat is this and why explain in simple. 22
11. DTO syntax bookmark_border Device tree source (DTS) format is a textual representation of a device tree (DT). The device tree compiler (DTC) processes this format into a binary DT, which is the form expected by the Linux kernel. Use references The DTC (device Tree compiler + overlay patches) project describes the DTS format in dtc-format.txt and manual.txt. DTO format and rules are described in dt-object-internal.txt. These documents describe how to update the main DT using node fragment@x and syntax __overlay__ in overlay DT. For example: / { fragment@0 { target = <&some_node>; __overlay__ { some_prop = "okay"; ... }; }; }; However, Google strongly recommends you do not use fragment@x and syntax __ov. 24
12. can you explain in simple and explain each code. 30
13. DTO syntax bookmark_border Device tree source (DTS) format is a textual representation of a device tree (DT). The device tree compiler (DTC) processes this format into a binary DT, which is the form expected by the Linux kernel. Use references The DTC (device Tree compiler + overlay patches) project describes the DTS format in dtc-format.txt and manual.txt. DTO format and rules are described in dt-object-internal.txt. These documents describe how to update the main DT using node fragment@x and syntax __overlay__ in overlay DT. For example: / { fragment@0 { target = <&some_node>; __overlay__ { some_prop = "okay"; ... }; }; }; However, Google strongly recommends you do not use fragment@x and syntax __ov. 33
14. Compile and verify bookmark_border You can use device tree compiler (DTC) to compile the device tree source (DTS) files. However, before applying the overlay device tree (DT) on the target main DT, you should also verify the result by simulating the behavior of the device tree overlay (DTO). Compile with DTC When using dtc to compile .dts, you must add option -@ to add a __symbols__ node in the resulting .dtbo. The __symbols__ node contains a list of all nodes that are marked with a label, which the DTO library can use for references. Sample command to build main DT .dts: dtc -@ -O dtb -o my_main_dt.dtb my_main_dt.dts Sample command to build the overlay DT .dts: dtc -@ -O dtb -o my_overlay_dt.dtbo my_overlay_dt.dts Note: If you encou. 37
15. Verify DTO results on the host Verification can help you identify errors that might occur when placing the overlay DT on the main DT. Before updating the target, you can verify the result of overlaying DT on the host by simulating the behavior of DTO using /include/ in .dts. Figure 1. Use syntax /include/ to simulate DTO on the host. Create a copy of the overlay .dts. In the copy, remove the first line header. Example: /dts-v1/; /plugin/; Save the file as my_overlay_dt_wo_header.dts (or any filename you want). Create a copy of the main .dts. In the copy, after the last line, append the include syntax for the file you created in step 1. For example: /include/ "my_overlay_dt_wo_header.dts" Save the file as my_main_dt_with_include.dts (or. 39
16. explain each code in simple words. 45
17. Figure 1. Use syntax /include/ to simulate DTO on the host. Create a copy of the overlay .dts. In the copy, remove the first line header. Example: /dts-v1/; /plugin/; Save the file as my_overlay_dt_wo_header.dts (or any filename you want). Create a copy of the main .dts. In the copy, after the last line, append the include syntax for the file you created in step 1. For example: /include/ "my_overlay_dt_wo_header.dts" Save the file as my_main_dt_with_include.dts (or any filename you want). Use dtc to compile my_main_dt_with_include.dts to get the merged DT, which should be the same result as DTO. For example: dtc -@ -O dtb -o my_merged_dt.dtb my_main_dt_with_include.dts Use dtc to dump my_merged_dt.dto. dtc -O dts -o my_merged_dt.dts my. 47
18. confusing. 53
19. Use multiple DTs bookmark_border Many SoC vendors and ODMs support the use of multiple device trees (DTs) on a device, enabling one image to power multiple SKUs or configurations. In such cases, the bootloader identifies the hardware and loads the corresponding DT at runtime: Figure 1. Multiple device tree overlays (DTOs) in bootloader. Note: Using multiple DTs isn't mandatory. Set up To add support for multiple DTs to the DTO model, set up a list of main DTs and another list of overlay DTs. Figure 2. Runtime DTO implementation for multiple DTs. The bootloader should be able to: Read the SoC ID and select the corresponding main DT. Read the board ID and select the set of overlay DTs accordingly. Only one main DT should be selecte. 56
20. 📂 Partition Format ------------------- If using `dtb` and `dtbo` partitions: * Use the **DT table format** (with headers and entries) * Each entry in `.dtbo` partition can be indexed * You reference overlays by **index** (like 0, 1, 2...)this is confusing. 59
21. ok. 61
22. explain this image. 62
23. DTB and DTBO partitions bookmark_border If your device tree blob (DTB) or device tree blob for overlay (DTBO) is in a unique partition, for example, the dtb and dtbo partition, use the following table structure and header format: Data structures The dt_table_header is only for the dtb/dtbo partition; you CAN'T append this format after the end of image.gz. If you have a single DTB or DTBO, you must still use this format (and the dt_entry_count in dt_table_header is 1). #define DT_TABLE_MAGIC 0xd7b7ab1e struct dt_table_header { uint32_t magic; // DT_TABLE_MAGIC uint32_t total_size; // includes dt_table_header + all dt_table_entry // and all dtb/dtbo uint32_t header_size; // s. 64

24. Optimize DTOs bookmark_border This page discusses optimizations you can make to your device tree overlay (DTO) implementation, describes restrictions against overlaying the root node, and details how to configure compressed overlays in the DTBO image. It also provides sample implementation instructions and code. Kernel command line The original kernel command line in device tree (DT) is located in the chosen/bootargs node. The bootloader must concatenate this location with other sources of kernel command line: /dts-v1/; / { chosen: chosen { bootargs = "..."; }; }; DTO cannot concatenate values from main DT and overlay DT, so you must put the kernel command line of the main DT in chosen/bootargs and the kernel command line of th. 70
25. libufdtexplain in simPLE WORDS. 76
26. What next should I read now. 78

1. Device tree overlays

bookmark_border
A device tree (DT) is a data structure of named nodes and properties that describe non-discoverable hardware. Kernels, such as the Linux kernel used in Android, use DTs to support a wide range of hardware configurations used by Android-powered devices. Hardware vendors supply their own device tree source (DTS) files, which are compiled into the device tree blob (DTB) file using the device tree compiler. These files are then used by the bootloader. The DTB file contains a binary-formatted flattened device tree.

A device tree overlay (DTO) enables a central DTB to be overlaid on the DT (a device tree blob for overlay (DTBO)). A bootloader using DTO can maintain the system-on-chip (SoC) DT and dynamically overlay a device-specific DT, adding nodes to the tree and making changes to properties in the existing tree.

Updates in Android 9 release
In Android 9, the bootloader must not modify the properties defined in the DTOs before passing the unified DTB to the kernel.

Here's a **simplified explanation** of **Device Tree Overlays (DTO)** and their role in Android (especially from Android 9 onwards):

✅ What is a Device Tree (DT)?

A **Device Tree (DT)** is like a hardware map written in a data format. It tells the Linux kernel about the hardware that is **not auto-discoverable**, like:

- GPIOs
- I2C/SPI devices
- PMICs
- Custom peripherals

It's structured as:

- **Nodes** (for hardware blocks)
- **Properties** (like address, interrupts)

🛠️ Files involved:

- **DTS (Device Tree Source):** Human-readable source file
- **DTB (Device Tree Blob):** Binary compiled from DTS using `dtc`
- **DTBO (Device Tree Blob Overlay):** A special binary that contains only changes to apply to a base DTB

🧩 What is a Device Tree Overlay (DTO)?

- A **Device Tree Overlay (DTO)** is a *partial Device Tree*.
- It **adds/modifies** nodes or properties to the base DTB.
- It allows reusing the same SoC-level DT and adding only **board-specific or peripheral-specific changes**.

Why it's useful:

- Modularity: Base SoC config + custom board config
- Reusability: One DTB for many devices
- Easier updates or feature toggling (e.g., enabling a camera or touchscreen)

🚀 How DTO Works in Android Boot:

1. Bootloader loads:
    - SoC-level DTB (base)
    - Board/device-level DTBO (overlay)
2. Bootloader merges them into a **unified DTB**

3. Passes the unified DTB to the Linux kernel

🔐 Changes in Android 9:

Before Android 9:

- Bootloader could change DTB/DTBO properties before passing to kernel.

After Android 9:

- **Bootloader must NOT modify** the DTO properties after merging.
- This enforces consistency and prevents runtime bugs due to tampered DT.

📂 File Locations in Android:

- **DTBs/DTBOs** are stored in:
    - `boot.img` (for legacy cases)
    - `dtbo.img` (dedicated image for overlays in newer Android)
- You define `BOARD_KERNEL_DTBO` in your `BoardConfig.mk`

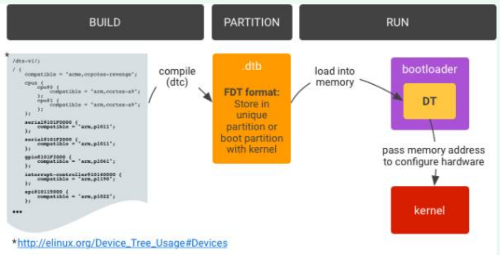
🧠 Summary:

Concept	Description
DTB	Binary file describing base hardware
DTO / DTBO	Patch to DTB for board/peripheral-specific updates
Use	Merge SoC and board-specific settings dynamically
Android 9 Rule	Bootloader cannot modify DTO before giving to kernel

---

Would you like a visual diagram showing how the base DTB and DTBO are merged by the bootloader?

2.


BUILD
PARTITION
RUN
compile
(dtc)
.dtb
FDT format:
Store in unique partition or boot partition with kernel
load into memory
bootloader
DT
pass memory address to configure hardware
kernel

```
*
/dts-v1/;
/ {
    compatible = "acme,coyotes-revenge";
    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
    serial@101F0000 {
        compatible = "arm,pl011";
    };
    serial@101F2000 {
        compatible = "arm,pl011";
    };
    gpio@101F3000 {
        compatible = "arm,pl061";
    };
    interrupt-controller@10140000 {
        compatible = "arm,pl190";
    };
    spi@10115000 {
        compatible = "arm,pl022";
    };
...
```

*http://elinux.org/Device_Tree_Usage#Devices

Load a DT
Loading a DT in bootloader involves building, partitioning, and running.

Great! Let's walk through the process shown in the **diagram** and explain **how a Device Tree (DT) is loaded** into the Android/Linux kernel during boot.

🔁 Full Flow: Loading a Device Tree (DT)

The diagram is divided into **three main stages**:

🏗️ 1. BUILD (Left Section)

- You start with a `.dts` **(Device Tree Source)** file.
- This is a text-based description of hardware: CPUs, GPIOs, UARTs, etc.
- You compile the `.dts` into a binary `.dtb` using the **Device Tree Compiler (dtc)**:

  ```bash
  dtc -I dts -O dtb my_board.dts -o my_board.dtb
  ```

💽 2. PARTITION (Middle Section)

- The resulting `.dtb` file is placed in:
  - A **dedicated partition** (e.g., `dtbo`, `vendor_boot`)
  - Or included in the **boot partition**
- Format used is **FDT** (Flattened Device Tree)
- It's just a compact, binary representation of the DT for fast parsing.

🚀 3. RUN (Right Section)

- **Bootloader** (e.g., U-Boot or Fastboot) loads the `.dtb` into memory.
- It passes a **pointer to the DT in memory** to the Linux kernel at boot time.
- The kernel reads this DT and configures all non-discoverable hardware accordingly:
  - It knows where UARTs are, how many cores to boot, etc.

✅ Summary of Key Terms:

Term	Meaning
`.dts`	Device Tree Source file (text format)
`.dtb`	Compiled DT blob (binary format)
`dtc`	Device Tree Compiler
FDT	Flattened Device Tree (in `.dtb`)
Bootloader	Loads DTB to memory and passes to kernel

Term	Meaning
Kernel	Uses DT to initialize hardware

✨ Extra Note on Device Tree Overlays (DTO):

While this diagram shows the basic **single DTB flow**, in **modern Android** (especially with **Project Treble** and Android 9+), the bootloader can:

- Load a **base DTB**
- Load one or more **DTBOs** (Device Tree Overlays)
- **Merge** them before passing the final DT to the kernel

This enables modularity — a single DTB per SoC and overlays for different boards/devices.

Let me know if you want the **extended diagram** including DTBO/DTO merging, or how `dtbo.img` works.

3.

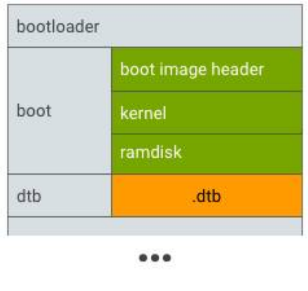


Figure 1. Typical implementation for loading device tree in bootloader.

To create and flash the DTB:

1a. Use the DTC (dtc) to compile DTS (.dts) into a DTB (.dtb). The DTB is formatted as a flattened DT. 1b. Flash the .dtb file into a bootloader runtime-accessible location (detailed below).

To partition, determine a bootloader runtime-accessible and trusted location in flash memory to put .dtb. Example locations:

Figure 2. Put .dtb in a boot partition by appending to image.gz and passing as "kernel" to mkbootimg.

Figure 3. Put .dtb in a unique partition (for example, dtb partition).

To load the DTB and start the kernel:

Load .dtb from storage into memory.
Start kernel using the memory address of the loaded DT.

Thanks for sharing the second image — this diagram explains how the **Device Tree Blob (DTB)** is stored and used during Android/Linux boot, showing **two ways to provide the DTB** to the kernel. Let's break it all down step by step:

🧱 Figure 1: Bootloader + Partitions Overview

The diagram shows these key partitions:

Partition	Contents
`bootloader`	The firmware that runs first and controls boot process
`boot`	Contains kernel + ramdisk + boot image header
`dtb`	A separate partition holding `.dtb`

⚙️ Steps: How to Use DTB in Android Boot

🔨 Step 1: Create the `.dtb`

- Use the Device Tree Compiler (DTC) to compile `.dts` into `.dtb`:

  bash

  ```
  dtc -I dts -O dtb board.dts -o board.dtb
  ```

💾 Step 2: Flash `.dtb` to Flash Memory

You have **two options** here:

📦 Option 1: Append `.dtb` to the boot image

- You **concatenate** the `.dtb` to the kernel image:

```bash
cat Image.gz board.dtb > Image.gz-dtb
```

- Then you use `mkbootimg` to create the full boot image:

```bash
mkbootimg --kernel Image.gz-dtb --ramdisk ramdisk.img --output boot.img
```

- So now, **boot.img contains the DTB** along with kernel and ramdisk.

🧩 Option 2: Use a separate `dtb` partition

- Instead of putting the DTB inside `boot.img`, flash it to a **dedicated partition**:

```bash
fastboot flash dtb board.dtb
```

- The bootloader knows to load the `.dtb` from the `dtb` partition.

🚀 Step 3: Load and Start Kernel

- Bootloader:
  - Loads the `.dtb` into memory
  - Loads the kernel and ramdisk
  - Passes the **memory address** of the DTB to the kernel via boot parameters
- Kernel:
  - Uses the DTB to initialize hardware (like CPUs, GPIOs, I2C, etc.)

✅ Summary Table:

Task	Option 1	Option 2
Store `.dtb`	Inside `boot.img` (appended to kernel)	Separate `dtb` partition
Tool to pack	`mkbootimg`	Just flash `.dtb` directly
Access at runtime	Bootloader extracts from `boot`	Bootloader reads from `dtb` partition
Flexibility	Less modular	More modular (e.g., used with DTO)

Would you like a combined diagram showing **DTB + DTBO (overlays)** and how they're merged before boot? This is important for Project Treble and dynamic hardware support.

4. Implement DTOs

bookmark_border
Implementing device tree overlays (DTOs) involves dividing the device tree (DT), building, partitioning, and running. After you have a working implementation, you must also maintain compatibility between the two DTs and determine a strategy for ensuring the security of each DT partition.

Divide the DT
Start by dividing the DT into two parts:

Main DT. The SoC-only part and the default configurations, provided by SoC vendor.
Overlay DT. The device-specific configurations, provided by ODM/OEM.
After dividing the DTs, you must ensure compatibility between main DT and overlay DT so that merging main DT and overlay DT results in a complete DT for the device. For details on DTO format and rules, see DTO syntax. For details on multiple DTs, see Use multiple DTs.

Build main and overlay DTs
To build the main DT:

Compile the main DT .dts into a .dtb file.
Flash the .dtb file into a bootloader runtime-accessible partition (detailed in [Partition DTs](#partition)).
To build the overlay DT:

Compile the overlay DT .dts into a .dtbo file. While this file format is the same as the .dtb file formatted as a flattened DT, the different file extension distinguishes it from the main DT.
Flash the .dtbo file into a bootloader runtime-accessible partition (detailed in [Partition DTs](#partition)).
For details on compiling with DTC and verifying DTO results on the host, see Compile and verify.

Here's a complete **simplified guide to implementing Device Tree Overlays (DTOs)** in Android/Linux, broken down step-by-step:

✅ 1. What Are You Implementing?

Device Tree Overlays (DTOs) allow you to **split the Device Tree** into:

- A **main DTB**: Contains SoC-level, default config (from SoC vendor)
- One or more **DTBOs**: Contains board-specific or peripheral-specific config (from OEM/ODM)

These are **merged by the bootloader** before the kernel boots.

🧩 2. Step-by-Step Implementation Process

🔹 Step 1: Divide the Device Tree

Split your `.dts` files like this:

- `main.dts` → Includes only SoC-wide config
  (CPU, RAM controller, generic buses, etc.)
- `overlay.dts` → Includes board-level config
  (panel, camera, GPIO, touch controller, etc.)

You **must ensure compatibility**:

- Nodes in `overlay.dts` must either add to or modify nodes in `main.dts`
- You can use `fragment@X` syntax in the overlay

🔹 Step 2: Compile the DTs

Compile Main DT:

```bash
dtc -I dts -O dtb -o main.dtb main.dts
```

Compile Overlay DT:

```bash
dtc -I dts -O dtb -o board.dtbo overlay.dts
```

💡 `.dtbo` is structurally the same as `.dtb`, just used for overlays.

🔹 Step 3: Partitioning – Store the DTs

Main DTB can be stored in:

- Boot partition (appended to kernel)
- Separate `dtb` partition

Overlay DTBO must go to:

- A dedicated `dtbo` partition (named `dtbo.img` by convention)

Use `fastboot` to flash:

```bash
fastboot flash dtb main.dtb
fastboot flash dtbo dtbo.img
```

🔹 Step 4: Merging and Running

At boot time:

1. **Bootloader loads** `main.dtb` and one or more `dtbo` entries from `dtbo.img`
2. It uses **libufdt** (in U-Boot, LK, etc.) to **merge overlays into the base DTB**
3. It passes the **merged DT** to the kernel

🧠 Android's bootloader uses `FDT` and `ufdt_apply_overlay()` to apply overlays.

🛡️ Security & Compatibility Tips

- ✅ **Secure each partition** (`boot`, `dtb`, `dtbo`) with verified boot
- ✅ **Ensure overlay doesn't break** base DT (e.g., mismatched node names or phandles)
- ✅ **Use `dtbo.img` format** when using Android's `mkdtboimg.py`

🛠️ Tooling

🔧 Build `.dtbo.img` from `.dtbo` files:

```bash
mkdtboimg.py create dtbo.img --dtbo board1.dtbo board2.dtbo --page_size 2048
```

🔍 Verify Overlay on Host:

```bash
ufdt_apply_overlay main.dtb board.dtbo merged.dtb
```

🗂️ Example Structure:

```css
device/
└── myboard/
    ├── dts/
    │   ├── main.dts
    │   └── overlay.dts
    ├── dtb/
    │   ├── main.dtb
    │   └── board.dtbo
    └── images/
        ├── boot.img
        ├── dtbo.img
```

Would you like an example `overlay.dts` fragment showing how to override or add properties?

> 5. ↪
> dtc -I dts -O dtb -o main.dtb main.dts
> explain

Absolutely! Let's break down the command you mentioned:

🔧 Command:

```bash
dtc -I dts -O dtb -o main.dtb main.dts
```

This command compiles a **Device Tree Source (.dts)** file into a **Device Tree Blob (.dtb)** file using the **Device Tree Compiler (dtc)**.

🧠 Meaning of Each Part:

Part	Meaning
`dtc`	**Device Tree Compiler** — the tool that compiles or decompiles device trees
`-I dts`	**Input format** is **`dts`** (Device Tree Source — a human-readable **`.dts`** file)
`-O dtb`	**Output format** is **`dtb`** (Device Tree Blob — binary format for the kernel)
`-o main.dtb`	Output file name will be **`main.dtb`**
`main.dts`	Input source file you want to compile

📁 File Types:

File	Description
`main.dts`	Human-readable text file describing hardware layout
`main.dtb`	Flattened, binary format used by the bootloader and kernel

---

📌 Example:

Suppose you have a **`main.dts`** like:

```dts
/ {
    compatible = "qcom,sm8350";
    model = "My Custom Board";

    memory@80000000 {
        device_type = "memory";
        reg = <0x80000000 0x40000000>; // 1GB RAM
    };

    chosen {
        bootargs = "console=ttyMSM0,115200n8";
    };
};
```

After running the command:

```bash
dtc -I dts -O dtb -o main.dtb main.dts
```

- It will generate **`main.dtb`**, a binary file that can be loaded by the bootloader.
- The **kernel will use it** to understand the board layout and initialize devices.

Would you also like the reverse command to convert `.dtb` back into `.dts`?

6. ⤷
U-Boot
what is this

Great — let's explain **U-Boot** in simple and clear terms:

🚀 What is U-Boot?

U-Boot (short for **Universal Bootloader**) is an open-source bootloader **used in embedded Linux systems**, including Android devices.

It’s the **first program** that runs after the device powers on (after firmware/ROM or minimal pre-loader). It is responsible for:

- Initializing the CPU, RAM, and peripherals
- Loading and passing the **Linux kernel**, **Device Tree (DTB)**, and optionally the **initramfs** (ramdisk)
- Setting up boot arguments
- Optionally loading overlays like **DTBO**
- Starting the kernel

🧱 Key Responsibilities of U-Boot

Task	Description
Initialize hardware	Sets up DDR, clocks, serial ports
Load kernel	Loads the Linux kernel binary (usually from storage like eMMC, SD card, or network)
Load DTB/DTBO	Loads and merges device trees and overlays (if any)
Pass parameters	Supplies kernel with bootargs (e.g. console settings, root filesystem path)
Run scripts	Executes boot commands (`bootcmd`) or environment variables
Optional interaction	User can stop boot and run commands manually (via serial/UART console)

📦 Example Flow in U-Boot

```bash
# Example U-Boot commands
load mmc 0:1 0x82000000 zImage         # Load kernel
load mmc 0:1 0x83000000 myboard.dtb    # Load DTB
bootz 0x82000000 - 0x83000000          # Boot the kernel with DTB
```

📁 Typical Partitions Used with U-Boot

Partition	Used for
`boot`	Contains kernel and ramdisk
`dtb`	(optional) DTB file if not included in `boot`
`dtbo`	(optional) Device Tree Overlays
`recovery`, `system`, etc.	Android partitions

🧠 Why Is U-Boot Important?

U-Boot is **flexible, portable, and highly customizable**. It plays a key role in:

- Early hardware initialization
- Deciding what and how to boot

- Enabling advanced features like:
    - Fastboot, TFTP booting, OTA update hooks
    - Secure Boot / Verified Boot
    - Multistage booting with SPL (Secondary Program Loader)

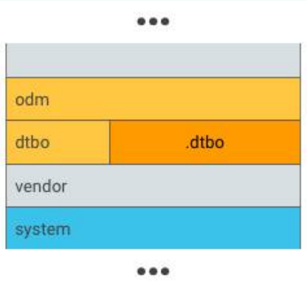
🔧 Developer Note

In Android, you may use U-Boot to:

- Load `boot.img` (contains kernel, ramdisk, maybe DTB)
- Load `dtbo.img` separately and apply overlays
- Use `mkimage`, `mkbootimg`, or `mkdtboimg.py` tools to prepare binaries

Would you like to see an actual `U-Boot` environment setup or example `boot.cmd` used to boot an Android device?

7.



Partition DTs
Determine a bootloader runtime-accessible and trusted location in flash memory to put .dtb and .dtbo.

Example locations for the main DT:

Part of boot partition, appended to the kernel (image.gz)
Separate DT blobs (.dtb) in dedicated partition (dtb)
Example locations for the overlay DT:

Thanks! The image and text explain **how to partition and store DTB and DTBO files** in Android devices. Let's simplify it step by step:

📦 Partitioning DTs (Device Trees) in Android

You need to store:

- **Main Device Tree** (`.dtb`)
- **Overlay Device Tree** (`.dtbo`)

These must be placed in **runtime-accessible** and **trusted locations**, so the **bootloader** can find and load them during boot.

🧩 1. Main DTB (SoC/Base)

Option	Description
✅ **Boot partition**	Append `.dtb` to the end of the **kernel image** (e.g., `Image.gz + .dtb`) and include it in **`boot.img`**. This is common in Android.
✅ **Separate `dtb` partition**	Store `.dtb` in a dedicated flash partition called **`dtb`**. Bootloader loads it separately.

📌 *Purpose*: Contains SoC-level hardware info (CPU, RAM, buses, etc.)

🧩 2. Overlay DTBO (Board/Peripheral Config)

Partition	Description
✅ **dtbo partition**	Store `.dtbo` files (device tree overlays) here. This is standard for Android Treble devices.
✅ **ODM partition** (optional)	ODM-specific DTBOs can also be stored here for board-level customizations.

📌 *Purpose*: Describes hardware differences — e.g., panel type, touchscreen, GPIO configuration

📊 Example Partition Table (from image)

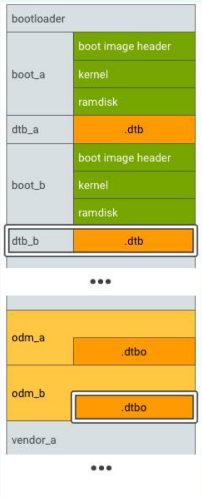
Partition	Purpose
`odm`	Optional vendor/device customization, including `.dtbo`
`dtbo`	Standard location for `.dtbo` overlays
`vendor`	Vendor-supplied HALs and binaries
`system`	Android framework and system apps

📌 Summary: Where to Store Device Tree Files

File Type	Typical Location
`.dtb` (main)	Appended to kernel in **`boot.img`** OR stored in **`dtb`** partition
`.dtbo` (overlay)	Stored in **`dtbo`** partition OR **`odm`** partition

---

Would you like an example of how to **create a `dtbo.img`** with multiple `.dtbo` files using **`mkdtboimg.py`**?

8. 

For devices that support seamless (A/B) updates, A/B the main DT and overlay DT partitions:

This image illustrates how **Device Tree (DTB)** and **Device Tree Overlays (DTBO)** are handled on **A/B partitioned Android devices** (used for **seamless updates**).

Let's break it down clearly:

🧩 What is A/B Partitioning?

A/B partitions provide two sets of partitions (`_a` and `_b`) so that:

- The device **boots from one slot** (say, `_a`)
- And **updates the other slot** (say, `_b`) in the background
- After a reboot, it can switch to the updated slot

Used for:
✅ Seamless system updates
✅ Rollback in case of failure

📦 Partitioning DTBs and DTBOs in A/B Devices

Component	A Slot	B Slot
`boot`	`boot_a`	`boot_b`
`dtb`	`dtb_a`	`dtb_b`
`odm`	`odm_a`	`odm_b`

📌 Key Details from Image:

🔶 Main DTB (`.dtb`)

- Stored separately as:
  - `dtb_a` for slot A
  - `dtb_b` for slot B
- These are matched with `boot_a` and `boot_b`

🔶 Overlay DTBO (`.dtbo`)

- Stored either in:

  - `odm_a` / `odm_b` partitions, or
  - dedicated `dtbo_a` / `dtbo_b` partitions on some devices

🚀 Boot Process (A/B Example):

Let's say **device is booting slot A**:

1. Bootloader reads `boot_a`, `dtb_a`, and optionally `odm_a`
2. Loads:
   - Kernel from `boot_a`
   - DTB from `dtb_a`
   - DTBO from `odm_a`
3. **Applies DTBO onto DTB**
4. Passes merged Device Tree to the kernel

If the update targets slot B:

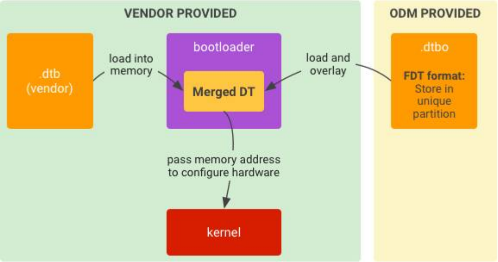
- New kernel → `boot_b`
- New DTB → `dtb_b`
- New DTBO → `odm_b`
- These are only activated once boot slot is switched

✅ Summary: A/B DT and DTO Partition Strategy

Component	Purpose	Partition
Kernel + Ramdisk	OS boot code	`boot_a`, `boot_b`
DTB	Base SoC-level hardware config	`dtb_a`, `dtb_b`
DTBO	Overlay for board-specific or peripheral configs	`odm_a`, `odm_b` (or separate `dtbo_*`)

Would you like a working example of how the **bootloader applies the overlay using `libufdt`**?

9.



Run in bootloader

Figure 5. Typical runtime implementation for DTO in bootloader.

Load .dtb from storage into memory.
Load .dtbo from storage into memory.
Overlay .dtb with .dtbo to be a merged DT.
Start kernel given the memory address of the merged DT.
Maintain compatibility
The main DTB (from SoC vendor) is treated as an API surface for DTBOs. After separating the DT into a SoC-common part and a device-specific part, you must keep the two parts mutually compatible in the future, including:

DT definition in main DT. For example, nodes, properties, labels. Any definition change in main DT could trigger changes in overlay DT. For example, to correct a node name in main DT, define an "alias" label that maps to the original node name (to avoid the change of overlay DT).
Overlay DT store location. For example, partition name, store format.
Ensure security
Bootloader must ensure the DTB or DTBO is secure, unmodified, and uncorrupted. You could use any solution to secure DTB or DTBO, for example, Boot image signature in VBoot 1.0 or AVB HASH footer (VBoot 2.0).

If DTB or DTBO is in a unique partition, you can add that partition to the trust chain of AVB. The trust chain starts from a hardware-protected root of trust and goes to the bootloader, which verifies the integrity and authenticity of DTB or DTBO partition.
If DTB or DTBO is in an existing partition (such as the odm partition), that partition should be in the trust chain of AVB. (DTBO partition could share a public key with odm partition).
For details, refer to Verified Boot.

This image and explanation outline the **runtime process for Device Tree Overlays (DTO)** on Android using the bootloader, with an emphasis on **security** and **partition strategy**. Let's break it down simply:

✅ What Happens at Runtime (Bootloader Stage)

📥 Step-by-Step Flow:

Step	Action
1. Load `.dtb`	The bootloader loads the **base DTB** from a vendor-supplied partition (e.g., **`dtb_a`**, **`boot`**, or a dedicated **`dtb`** partition).
2. Load `.dtbo`	The bootloader then loads the **overlay `.dtbo`** from an ODM-supplied partition (e.g., **`dtbo`**, **`odm_a`**, etc.).
3. Merge DT	The bootloader **applies the overlay (`.dtbo`) to the base DTB** using a tool like **`libufdt`**, creating a **Merged DT** in memory.
4. Pass to kernel	It passes the memory address of the **Merged DT** to the Linux kernel for booting.

🧱 Who Provides What?

Provider	Supplies	Example Partitions
Vendor (SoC provider)	**`.dtb`** (SoC and base hardware info)	**`boot`**, **`dtb_a`**, **`dtb_b`**
ODM/OEM (board maker)	**`.dtbo`** (board or SKU-specific config)	**`dtbo`**, **`odm_a`**, **`odm_b`**

🔄 Maintain Compatibility

Since `.dtbo` is applied **on top of** `.dtb`, `.dtb` **becomes an API surface**:

- Changes to `.dtb` (like renaming nodes, removing labels) **can break** `.dtbo`
- To safely evolve `.dtb`:
    - Use `labels` or `aliases` in `.dts` to maintain compatibility
    - Don't remove or rename nodes used in overlays

📝 *Example:*

```dts
serial0: serial@9000000 {
    compatible = "qcom,uart";
    ...
};
// later in .dtbo
fragment@0 {
    target = <&serial0>;
    ...
}
```

🔐 Ensure Security of DTB and DTBO

Security Concern	Recommended Solution
Unauthorized modification	Use **Verified Boot (VBoot)**
Partition integrity	Use **AVB (Android Verified Boot)** to protect `dtb`, `dtbo`, or `odm` partitions
Signature enforcement	Add DTB/DTBO partitions to **AVB trust chain**, so bootloader verifies them

✅ AVB Trust Chain:

Starts at **hardware root-of-trust** → **bootloader** → **partitions** (`dtb`, `dtbo`, `odm`, `boot`)

🧠 Summary

Component	Role	Runtime Action
`.dtb`	Base Device Tree	Loaded from vendor partition
`.dtbo`	Overlay Device Tree	Loaded from ODM partition
Bootloader	Combines both using `libufdt`	Merges `.dtb` + `.dtbo` into Merged DT
Kernel	Uses merged DT to configure hardware	Receives memory address of Merged DT

Would you like:

1. An example `overlay.dts` file?
2. The command to compile `.dtbo` files and build a `dtbo.img`?
3. An AVB configuration snippet to protect `dtbo` partition?

10. ↪
AVB Trust Chain
what is this and why explain in simple

Great question! Let's break down the **AVB Trust Chain** in **very simple words**, especially in the context of `.dtb` and `.dtbo`.

🔐 What is AVB?

AVB = **Android Verified Boot**
It's a security system that **ensures all software loaded at boot is trusted and not tampered with**.

🧱 What is the AVB Trust Chain?

Imagine booting your phone is like **climbing a ladder**, where each step **verifies** the next step before trusting it.

This chain is called the **AVB trust chain**:

🔗 Trust Chain Order (Simple View):

1. ✅ **Hardware Root of Trust**
2. ⬇️ verifies → **Bootloader**
3. ⬇️ verifies → **DTB** (`.dtb`)
4. ⬇️ verifies → **DTBO** (`.dtbo`)
5. ⬇️ verifies → **Kernel + Ramdisk**
6. ⬇️ verifies → **System, Vendor, ODM partitions**

Each "link" in the chain checks the **digital signature or hash** of the next "link" before running it.

📦 Why Does This Matter for `.dtb` and `.dtbo`?

- `.dtb` (device tree blob) describes hardware — it can **enable or disable devices**.
- `.dtbo` (device tree overlay) can **modify critical configs** like GPIOs, power, panel, touchscreen, etc.

❗ If someone tampers with `.dtb` or `.dtbo`, they could:

- Trick the kernel into booting on unsupported hardware
- Enable debugging backdoors
- Break safety-critical systems (e.g., voltage regulators)

✅ How Does AVB Help?

AVB ensures that:

- The bootloader will **only load `.dtb` and `.dtbo` if they are cryptographically verified**
- Any mismatch in the **hash** or **signature** means the boot will stop or fall back to recovery

So, **no unsigned or malicious `.dtbo` can run**, keeping your device safe.

🧠 Summary in Simple Words:

Term	Meaning
AVB	Protects Android boot process from tampering

Term	Meaning
Trust Chain	Step-by-step verification from hardware → bootloader → kernel → system
Why important for `.dtb/.dtbo`	Because they control hardware setup. If modified, can break or hijack the system
How protected	Via **signatures or hashes**, verified before loading

---

Let me know if you'd like a simple visual diagram or example `avbtool` command that adds `.dtbo` to the trust chain.

11. DTO syntax

bookmark_border
Device tree source (DTS) format is a textual representation of a device tree (DT). The device tree compiler (DTC) processes this format into a binary DT, which is the form expected by the Linux kernel.

Use references
The DTC (device Tree compiler + overlay patches) project describes the DTS format in dtc-format.txt and manual.txt. DTO format and rules are described in dt-object-internal.txt. These documents describe how to update the main DT using node fragment@x and syntax __overlay__ in overlay DT. For example:

```
/ {
 fragment@0 {
  target = <&some_node>;
   __overlay__ {
    some_prop = "okay";
    ...
   };
 };
};
```

However, Google strongly recommends you do not use fragment@x and syntax __overlay__, and instead use the reference syntax. For example:

```
&some_node {
 some_prop = "okay";
 ...
};
```

Reference syntax is compiled by dtc into the same object as the above using syntax __overlay__. This syntax doesn't force you to number the fragments, enabling you to read and write overlay DTS easily. If your dtc doesn't support this syntactic sugar, use the dtc in AOSP.

Use labels
To allow undefined references to nodes not present at compilation time, the overlay DT .dts file must have a tag /plugin/ in its header. For example:

```
/dts-v1/;
/plugin/;
```

From here you can target the nodes to be overlaid using a reference, which is an absolute node path prefixed with an ampersand (&). For example, for node@0 in the main DT:

Define labels in the main DT ... ... then use the labels.

[my_main_dt.dts]

```
/dts-v1/;

/ {
 my_node: node@0 {
  status = "disabled";

  my_child: child@0 {
   value = <0xffffffff>;
  };
 };
};
```

[my_overlay_dt.dts]

```
/dts-v1/;
/plugin/;

&my_node {
  status = "okay";
};

&my_child {
  value = <0x1>;
};
```

Override

If the reference target property exists in the main DT, it is overridden after DTO; otherwise, it is appended. For example:

main.dts overlay.dts Merged Result

```
[my_main_dt.dts]

/dts-v1/;

/ {
  compatible = "corp,foo";

  my_node: node@0 {
    status = "disabled";
  };
};

[my_overlay_dt.dts]

/dts-v1/;
/plugin/;

&my_node {
  status = "okay";
};

/dts-v1/;

/ {
  compatible = "corp,foo";

  ...

  node@0 {
    linux,phandle = <0x1>;
    phandle = <0x1>;
    status = "okay";
  };
};
```

Append

If the reference target property doesn't exist in the main DT, it is appended after DTO. For example:

main.dts overlay.dts Merged Result

```
[my_main_dt.dts]

/dts-v1/;

/ {
  compatible = "corp,foo";

  my_node: node@0 {
    status = "okay";
  };
};

[my_overlay_dt.dts]

/dts-v1/;
/plugin/;
```

```
&my_node {
  new_prop = "bar";
};

/dts-v1/;

/ {
  compatible = "corp,foo";

  ...

  node@0 {
    linux,phandle = <0x1>;
    phandle = <0x1>;
    status = "okay";
    new_prop = "bar";
  };
};
```

Child nodes

Examples of child node syntax:

main.dts   overlay.dts   Merged Result

```
[my_main_dt.dts]

/dts-v1/;

/ {
  compatible = "corp,foo";

  my_nodes: nodes {
    compatible = "corp,bar";

    node@0 {
      status = "disabled";
    };
  };
};

[my_overlay_dt.dts]

/dts-v1/;
/plugin/;

&my_nodes {
  new_prop1 = "abc";

  node@0 {
    status = "okay";
    new_prop2 = "xyz";
  };
};

/dts-v1/;

/ {
  compatible = "corp,foo";

  ...

  nodes {
    linux,phandle = <0x1>;
    phandle = <0x1>;
    compatible = "corp,bar";
    new_prop1 = "abc";
  node@0 {
   linux,phandle = <0x2>;
   phandle = <0x2>;
   status = "okay";
   new_prop2 = "xyz";
  };
 };
};
```

Let’s break down the **Device Tree Overlay (DTO) syntax** in **simple, clear terms** with examples so you can understand it easily.

🧠 What Is DTO Syntax?

Device Tree Source (DTS) is a human-readable file that describes hardware.

A **Device Tree Overlay (DTO)** is a way to **modify or extend** the base device tree at boot time — without changing the original `.dts`.

🛠️ How DTO Works (Basic Idea):

- You write an overlay `.dts` file
- The compiler converts it into `.dtbo` (overlay binary)
- The bootloader applies it on top of the `.dtb`
- The result is one merged tree used by the kernel

🔡 DTO Syntax Basics

✅ 1. Use `/plugin/` Tag in Overlay

This tells the compiler it's an overlay:

```dts
/dts-v1/;
/plugin/;
```

✅ 2. Use `&label` to Target Nodes

You don’t write the full node path — just refer to a **label** already defined in the main `.dts`.

Example: Main DT has labels

```dts
/ {
  my_node: node@0 {
    status = "disabled";
  };
};
```

Overlay DT refers to it:

```dts
/ {
  &my_node {
    status = "okay";
  };
};
```

💡 `&my_node` is a label reference — this avoids hardcoding full node paths.

✅ 3. Overriding vs Appending

Case	What Happens
Property **exists**	It is **overridden**
Property **doesn’t exist**	It is **appended**

Overriding:

```
dts

// base.dts
my_node: node@0 {
  status = "disabled";
};

// overlay.dts
&my_node {
  status = "okay";   // overrides previous
}
```

Appending:

```
dts

// base.dts
my_node: node@0 {
  status = "okay";
};

// overlay.dts
&my_node {
  new_prop = "bar";  // appended
}
```

✅ 4. Modifying Child Nodes

You can change subnodes (children) like this:

```
dts

// main.dts
my_nodes: nodes {
  node@0 {
    status = "disabled";
  };
};

// overlay.dts
&my_nodes {
  node@0 {
    status = "okay";
  };
}
```

You can also add properties to parent **and** children:

```
dts

&my_nodes {
  new_prop = "hello";
  node@0 {
    new_child_prop = "value";
  };
}
```

🛠️ DTO Syntax Styles

❌ Old Style (fragment@x + overlay)

```
dts

fragment@0 {
  target = <&some_node>;
  __overlay__ {
    some_prop = "okay";
  };
};
```

🚫 Google **recommends NOT using** this anymore (harder to read/maintain).

✅ New Style (Preferred)

```
dts

&some_node {
  some_prop = "okay";
};
```

✅ This is cleaner, easier to read, and does the same thing!

🧪 Compilation Tip

Compile overlay like:

```
bash

dtc -I dts -O dtb -o my_overlay.dtbo my_overlay.dts
```

✅ Summary Table

Feature	Syntax Example	Description
Target a node	`&my_node`	Refers to label in main DT
Override property	`status = "okay";`	Replaces existing property
Add new property	`new_prop = "abc";`	Appended if not in base
Update children	`node@0 { ... };`	Modifies subnodes
Required tags	`/plugin/;`	Needed in overlay DT

Would you like a **real-world overlay.dts example** that enables or disables a hardware peripheral (like UART or I2C)?

12. can you explain in simple and explain each code

Absolutely! Let’s go step by step to understand **Device Tree Overlay (DTO) syntax** in very simple language — and I’ll explain every code block for you.

🧠 What is a Device Tree Overlay (DTO)?

A **Device Tree (DT)** describes the hardware of your device to the Linux kernel (like CPU, memory, UART, I2C, etc.).

A **Device Tree Overlay (DTO)** is a **patch** that **modifies or adds to the original device tree at boot time**, without editing the original `.dts` file.

It’s useful when:

- You want to change hardware settings
- You support multiple boards with minor differences
- You want to enable or disable peripherals (like turning UART on/off)

✅ Basic DTO Structure

Code:

```
dts

/dts-v1/;
/plugin/;

&my_node {
  status = "okay";
};
```

Explanation:

Line	Meaning
`/dts-v1/;`	This marks the file as a Device Tree Source version 1
`/plugin/;`	Tells the compiler this file is an **overlay**, not a full DT
`&my_node {}`	Refers to a node **already defined** in the main device tree using a **label**
`status = "okay";`	Overrides or adds the **`status`** property to that node (turns the device ON)

🏷️ Example 1: Label Reference and Override

Main DT (`main.dts`)

```
dts

/ {
  my_uart: serial@9000000 {
    status = "disabled";
  };
};
```

Overlay DT (`overlay.dts`)

```
dts

/dts-v1/;
/plugin/;

&my_uart {
  status = "okay";
};
```

Explanation:

- `my_uart` is a label in the main DT
- Overlay turns the UART device **ON** by setting `status = "okay"`
- Before: `disabled` (off)
- After: `okay` (on)

🆕 Example 2: Appending a New Property

Main DT:

```
dts

my_device: device@0 {
  status = "okay";
};
```

Overlay:

```
dts

/dts-v1/;
/plugin/;

&my_device {
  new_prop = "hello";
};
```

Explanation:

- `new_prop` did **not exist** before
- Overlay **adds** `new_prop = "hello";`
- Final node:

```
dts

device@0 {
  status = "okay";
  new_prop = "hello";
};
```

👶 Example 3: Editing a Child Node

Main DT:

```
dts

my_nodes: nodes {
  node@0 {
    status = "disabled";
  };
};
```

Overlay:

```
dts

/dts-v1/;
/plugin/;

&my_nodes {
  node@0 {
    status = "okay";
    new_prop = "xyz";
  };
};
```

Explanation:

- `node@0` is a **child** inside `my_nodes`
- You change `status` and add `new_prop`
- Result:

```
dts

nodes {
  node@0 {
    status = "okay";
    new_prop = "xyz";
  };
};
```

⚠️ Old Style (Not Recommended)

```
dts

fragment@0 {
  target = <&my_node>;
  __overlay__ {
    status = "okay";
  };
};
```

Why avoid?

- Hard to read
- Hard to maintain
- New syntax (`&my_node {}`) is much cleaner

✅ Summary

Action	Syntax	What it does
Override property	`status = "okay";`	Changes a value
Add new property	`new_prop = "value";`	Appends new
Change child node	`child@0 { ... }`	Modify subnodes
Target existing node	`&label_name {}`	Uses a label from main DT
Required for DTO	`/plugin/;`	Marks this as an overlay

Would You Like a Full Example?

For example, a DTO that:

- Enables UART
- Sets baud rate
- Adds a GPIO

Let me know and I'll write a full `.dts` + `.dtbo` + `dtc` compile steps!

13. DTO syntax

bookmark_border
Device tree source (DTS) format is a textual representation of a device tree (DT). The device tree compiler (DTC) processes this format into a binary DT, which is the form expected by the Linux kernel.

Use references
The DTC (device Tree compiler + overlay patches) project describes the DTS format in dtc-format.txt and manual.txt. DTO format and rules are described in dt-object-internal.txt. These documents describe how to update the main DT using node fragment@x and syntax __overlay__ in overlay DT. For example:

```
/ {
  fragment@0 {
    target = <&some_node>;
      __overlay__ {
        some_prop = "okay";
        ...
      };
  };
};
```

However, Google strongly recommends you do not use fragment@x and syntax __overlay__, and instead use the reference syntax. For example:

```
&some_node {
  some_prop = "okay";
  ...
};
```

Reference syntax is compiled by dtc into the same object as the above using syntax __overlay__. This syntax doesn't force you to number the fragments, enabling you to read and write overlay DTS easily. If your dtc doesn't support this syntactic sugar, use the dtc in AOSP.

Use labels
To allow undefined references to nodes not present at compilation time, the overlay DT .dts file must have a tag /plugin/ in its header. For example:

```
/dts-v1/;
/plugin/;
```

From here you can target the nodes to be overlaid using a reference, which is an absolute node path prefixed with an ampersand (&). For example, for node@0 in the main DT:

Define labels in the main DT ... ... then use the labels.

[my_main_dt.dts]

```
/dts-v1/;

/ {
  my_node: node@0 {
    status = "disabled";

    my_child: child@0 {
      value = <0xffffffff>;
    };
  };
};
```

[my_overlay_dt.dts]

```
/dts-v1/;
/plugin/;

&my_node {
  status = "okay";
};

&my_child {
  value = <0x1>;
};
Override
If the reference target property exists in the main DT, it is overridden after DTO; otherwise, it is appended. For example:

main.dts	overlay.dts	Merged Result

[my_main_dt.dts]

/dts-v1/;

/ {
  compatible = "corp,foo";

  my_node: node@0 {
    status = "disabled";
  };
};

[my_overlay_dt.dts]

/dts-v1/;
/plugin/;

&my_node {
  status = "okay";
};

/dts-v1/;

/ {
  compatible = "corp,foo";

  ...

  node@0 {
    linux,phandle = <0x1>;
    phandle = <0x1>;
    status = "okay";
  };
};
Append
If the reference target property doesn't exist in the main DT, it is appended after DTO. For example:

main.dts	overlay.dts	Merged Result

[my_main_dt.dts]

/dts-v1/;

/ {
  compatible = "corp,foo";

  my_node: node@0 {
    status = "okay";
  };
};

[my_overlay_dt.dts]

/dts-v1/;
/plugin/;
```

```
&my_node {
  new_prop = "bar";
};

/dts-v1/;

/ {
  compatible = "corp,foo";

  ...

  node@0 {
    linux,phandle = <0x1>;
    phandle = <0x1>;
    status = "okay";
    new_prop = "bar";
  };
};
```

Child nodes
Examples of child node syntax:

main.dts   overlay.dts   Merged Result

```
[my_main_dt.dts]

/dts-v1/;

/ {
  compatible = "corp,foo";

  my_nodes: nodes {
    compatible = "corp,bar";

    node@0 {
      status = "disabled";
    };
  };
};

[my_overlay_dt.dts]

/dts-v1/;
/plugin/;

&my_nodes {
  new_prop1 = "abc";

  node@0 {
    status = "okay";
    new_prop2 = "xyz";
  };
};

/dts-v1/;

/ {
  compatible = "corp,foo";

  ...

  nodes {
    linux,phandle = <0x1>;
    phandle = <0x1>;
    compatible = "corp,bar";
    new_prop1 = "abc";
    node@0 {
      linux,phandle = <0x2>;
      phandle = <0x2>;
      status = "okay";
      new_prop2 = "xyz";
    };
  };
};
```

Great! Let's now explain the **DTO syntax** and **each code example** from your message in very **simple and clear language**, with comments and reasoning.

🌳 What is a Device Tree Overlay (DTO)?

A **Device Tree (DT)** is like a **map of your hardware**, telling the Linux kernel what devices are present and how to use them (e.g., UART, I2C, GPIO).

A **Device Tree Overlay (DTO)** is like a **patch** to update or change parts of that map **at boot time** — without changing the original file.

🛠️ DTO Basics

Required at top of overlay file:

```dts
/dts-v1/;
/plugin/;
```

- **`/dts-v1/;`** → Required in all DT files (standard marker).
- **`/plugin/;`** → Tells the compiler this is an overlay that depends on the base DT.

🚫 Old Syntax (Not Recommended):

```dts
/ {
  fragment@0 {
    target = <&some_node>;
```

14. Compile and verify

bookmark_border
You can use device tree compiler (DTC) to compile the device tree source (DTS) files. However, before applying the overlay device tree (DT) on the target main DT, you should also verify the result by simulating the behavior of the device tree overlay (DTO).

Compile with DTC
When using dtc to compile .dts, you must add option -@ to add a __symbols__ node in the resulting .dtbo. The __symbols__ node contains a list of all nodes that are marked with a label, which the DTO library can use for references.

Sample command to build main DT .dts:

```
dtc -@ -O dtb -o my_main_dt.dtb my_main_dt.dts
```

Sample command to build the overlay DT .dts:

```
dtc -@ -O dtb -o my_overlay_dt.dtbo my_overlay_dt.dts
```

Note: If you encounter the DTC build error: invalid option --'@', you might need to update your DTC version. Upstream of AOSP, the official DTC supports DTO as of version 1.4.4 and most patches are merged after December 2016. For DTO support, we recommend using the external/dtc in AOSP, which is synced with the latest DTC (with DTO patches merged as needed).

Let's break this down in **very simple language** and explain **how to compile and verify a Device Tree and its Overlay (DTO)** using `dtc`.

🧰 What is `dtc`?

`dtc` = Device Tree Compiler

It converts human-readable `.dts` files (text) into `.dtb` or `.dtbo` files (binary) that the kernel and bootloader can use.

🧪 Why Verify Before Boot?

Before you apply the **overlay (.dtbo)** onto the **main DT (.dtb)** on a real device, it's smart to **test the merge** on your PC — this helps you catch errors early.

But first, you need to compile both files correctly.

✅ Step 1: Compile the Main DT `.dts` File

```bash
dtc -@ -O dtb -o my_main_dt.dtb my_main_dt.dts
```

✅ Explanation:

Part	Meaning
`-@`	Includes a special node called **`__symbols__`** in the **`.dtb`**, which stores all label names. DTO needs this to work properly.
`-O dtb`	Output format = Device Tree Blob (binary)
`-o my_main_dt.dtb`	Output file name
`my_main_dt.dts`	Your main device tree source file

✅ Step 2: Compile the Overlay `.dts` File

```bash
dtc -@ -O dtb -o my_overlay_dt.dtbo my_overlay_dt.dts
```

✅ Explanation:

Same as above, but the output is a `.dtbo` (device tree **overlay**).

`.dtbo` = compiled overlay
`.dtb` = compiled base/main tree

⚠️ Common Error:

```text
dtc: invalid option -- '@'
```

✅ Cause:

Your version of `dtc` is too old and **doesn't support overlays** (missing `-@` support).

🛠️ Fix:

Use the `dtc` from **AOSP** (Android source code):

```bash
# Inside AOSP tree
cd external/dtc
mm   # or use: make dtc

# or build standalone
make -C external/dtc
```

That version includes all patches for DTO (Device Tree Overlay), including `-@`.

🧪 Bonus: How to Verify the Merged Tree

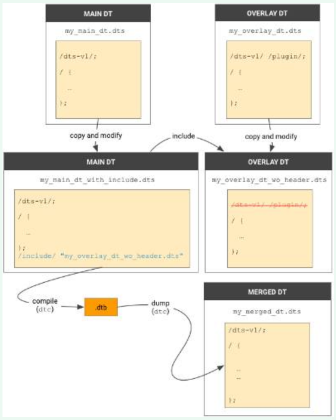
Once compiled, you can test the merge using tools like `ufdt_apply_overlay` (from AOSP's libufdt) or other verification scripts.

✅ Summary

Task	Command
Compile base DT	`dtc -@ -O dtb -o main.dtb main.dts`
Compile overlay DT	`dtc -@ -O dtb -o overlay.dtbo overlay.dts`
Error with `-@`	Use updated `dtc` from AOSP (`external/dtc`)
Why `-@` is needed	To include `__symbols__` (used to resolve label references in overlay)

Would you like me to show a **real full example**: main.dts, overlay.dts, and how to merge + inspect the result?

15.



Verify DTO results on the host
Verification can help you identify errors that might occur when placing the overlay DT on the main DT. Before updating the target, you can verify the result of overlaying DT on the host by simulating the behavior of DTO using /include/ in .dts.

Figure 1. Use syntax /include/ to simulate DTO on the host.

Create a copy of the overlay .dts. In the copy, remove the first line header. Example:

```
/dts-v1/;
/plugin/;
```

Save the file as my_overlay_dt_wo_header.dts (or any filename you want).
Create a copy of the main .dts. In the copy, after the last line, append the include syntax for the file you created in step 1. For example:

```
/include/ "my_overlay_dt_wo_header.dts"
```

Save the file as my_main_dt_with_include.dts (or any filename you want).
Use dtc to compile my_main_dt_with_include.dts to get the merged DT, which should be the same result as DTO. For example:

```
dtc -@ -O dtb -o my_merged_dt.dtb my_main_dt_with_include.dts
```

Use dtc to dump my_merged_dt.dto.

```
dtc -O dts -o my_merged_dt.dts my_merged_dt.dtb
```

Verify DTO in Android 9
Android 9 requires a device tree blob overlay (DTBO) partition. To add nodes or make changes to the properties in the SoC DT, the bootloader must dynamically overlay a device specific DT over the SoC DT.

Indicate applied overlays
To enable the Vendor Test Suite (VTS) to assess the correctness of overlay app, vendors must add a new kernel command line parameter androidboot.dtbo_idx that indicates the overlays selected from the DTBO partition. In Android 12 using kernel version 5.10 or greater, this parameter passes through bootconfig. For example, the parameter androidboot.dtbo_idx=x,y,z reports x, y and z as the zero-based indices of the DTOs from the DTBO partition applied (in that order) by the bootloader to the base DT.

Overlays can apply to nodes from the main DT or add new nodes, but can't refer to a node added in a previous overlay. This restriction is necessary because the overlay app doesn't merge the overlay symbol table with the main DT symbol table (not merging avoids conflicts in symbol names and complication of dependencies between overlays).

Example: Invalid overlays
In this example, overlay_2.dts refers to node e , which was added by overlay_1.dts. After overlay_1 is applied to the main DT, if an attempt is made to apply overlay_2 to the resultant DT, the overlay app fails with an error that the symbol e isn't present in the symbol table for the base DT.

main.dts   overlay_1.dts   overlay_2.dts

[main.dts]

```
/dts-v1/;
```

```
/ {
  a: a {};
  b: b {};
  c: c {};
};

[overlay_1.dts]

/dts-v1/;
/plugin/;

&b { ref1 =  <&a>;
    e: e {
        prop = <0x0a>;
        phandle = <0x04>;
    };
};

[overlay_2.dts]

/dts-v1/;
/plugin/;

/* invalid! */
&e {
    prop = <0x0b>;
};
Example: Valid overlays
In this example, overlay_2.dts refers only to node b from the main DTS. When overlay_1 is applied to the base DT, then followed by the app of overlay_2, the value of property prop in node e (set by overlay_1.dts) is overwritten by the value set by overlay_2.dts.

main.dts        overlay_1.dts        overlay_2.dts

[final.dts]

/dts-v1/;

/ {
  a: a {};
  b: b {};
  c: c {};
};

[overlay_1.dts]

/dts-v1/;
/plugin/;


&b { ref1 =  <&a>;
     e {
          prop = <0x0c>;
      };
};

[overlay_2.dts]

/dts-v1/;
/plugin/;

/* valid */
&b { ref1 =  <&c>;
     e {
          prop = <0x0d>;
      };
};
```

Implement the DTBO partition
To implement the required DTBO partition, ensure the bootloader can do the following:

Identify the board it is running on and select the corresponding overlays to be applied.
Append the androidboot.dtbo_idx parameter to the kernel command line.
The parameter must indicate, the zero-based indices of the DTOs from the DTBO partition image it applied to the base DT (in the same order).
The indices must refer to the position of the overlay in the DTBO partition.
For details on the structure of the DTBO partition, refer to Device tree overlays.

Validate the DTBO partition
You can use VTS to verify the following:

Existence of the kernel command line parameter androidboot.dtbo_idx (by checking that Init has automatically set up the corresponding ro.boot.dtbo_idx system property).
Validity of the ro.boot.dtbo_idx system property (by checking that the property specifies at least one valid DTBO image index).
Validity of the DTBO partition (also verifies the overlays in the DTBO partition that are applied to the base DT).
Additional nodes or property changes in the resulting DT are presented to the Linux kernel.
For example, in the following overlays and final DT, adding androidboot.dtbo_idx=5,3 to the kernel command line passes validation but adding androidboot.dtbo_idx=3,5 to the kernel command line doesn't pass validation.

Overlay DT at index 3 Overlay DT at index 5

[overlay_1.dts]

```
/dts-v1/;
/plugin/;

&c { prop = <0xfe>; };
```

[overlay_2.dts]

```
/dts-v1/;
/plugin/;

&c { prop = <0xff>; };
```

Final DT

```
/dts-v1/;
/ {

	a {
		phandle = <0x1>;
	};

	b {
		phandle = <0x2>;
	};

	c {
		phandle = <0x3>;
		prop = <0xfe>;
	};

	__symbols__ {
		a = "/a";
		b = "/b";
		c = "/c";
	};
};
```

Great! Let's explain this entire **DTO simulation and verification process** in **simple terms**, breaking it into **small digestible steps** with explanations for each diagram and concept.

🧪 What Is This All About?

This is about **verifying Device Tree Overlay (DTO) behavior on your host machine** (PC or development machine) *before flashing to your device*. This lets you:

✅ Detect and fix errors early
✅ Avoid bricking or misconfiguring hardware
✅ See the exact merged result of `.dts + .dtbo`

📌 High-Level Steps

1. **Main DT** = `.dts` file describing common SoC hardware
2. **Overlay DT** = `.dts` describing board-specific or dynamic additions
3. Simulate DTO merge using `#include` and `dtc`
4. Dump the result and inspect

🧩 Step-by-Step with Diagrams

✅ Step 1: Start with Your DTS Files

You have:

- `my_main_dt.dts`
- `my_overlay_dt.dts` with `/plugin/` (standard for overlays)

```perl
Main:          Overlay:
-------------  ---------------------------
/dts-v1/;      /dts-v1/;
/ { … };       /plugin/;
/ { … };
```

✅ Step 2: Copy and Modify

➡️ You create a version of the **overlay without header lines**:

```dts
// my_overlay_dt_wo_header.dts
/* remove these lines:
/dts-v1/;
/plugin/;
*/
&some_node {
    some_prop = "okay";
};
```

➡️ Create a **new main .dts** that includes the overlay:

```dts
// my_main_dt_with_include.dts
/dts-v1/;

/ {
  ...your main DT here...
};

/include/ "my_overlay_dt_wo_header.dts"
```

✅ Step 3: Compile with `dtc`

```bash
dtc -@ -O dtb -o my_merged_dt.dtb my_main_dt_with_include.dts
```

This generates the **merged binary DT** (just like bootloader would do with `.dtbo` + `.dtb`).

✅ Step 4: Dump and Inspect

```bash
dtc -O dts -o my_merged_dt.dts my_merged_dt.dtb
```

You now get the final **merged** DT **in readable `.dts` format**.

✅ What You Achieved

You simulated the exact **overlay process** that the bootloader performs — but on your PC. This helps confirm:

- Overlay DT is valid
- It merges with base DT
- Resulting merged DT behaves as expected

🔐 `androidboot.dtbo_idx` — What is this?

In Android 9 and above, you must **tell the kernel which overlays you applied** using this kernel command line parameter.

```ini
androidboot.dtbo_idx=3,5
```

This means: “I applied overlay at index 3, then overlay at index 5 from the DTBO partition.”

Used by:

- Kernel
- Android’s Init system (**`ro.boot.dtbo_idx`**)
- VTS test tools

❌ Invalid DTO Example

```dts
// overlay_1.dts adds node "e"
&b {
  e: e {
    prop = <0x0a>;
  };
}

// overlay_2.dts tries to modify "e"
&e {
  prop = <0x0b>; // ❌ Invalid! Not part of main DT
}
```

This fails, because **you can’t reference a node added by a previous overlay**. Each overlay is applied independently over the base DT.

✅ Valid DTO Example

```dts
// overlay_1.dts
&b {
  e {
    prop = <0x0c>;
  };
}

// overlay_2.dts — same structure
&b {
  e {
    prop = <0x0d>; // ✅ Valid! Not referencing from overlay_1
    };
  }
```

Here, `e` is created fresh in both overlays, and overlay_2 simply **overwrites** what overlay_1 wrote.

✅ Summary Table

Task	Tool / Action
Simulate overlay	Use `/include/` with modified overlay
Compile merged DT	`dtc -@ -O dtb -o out.dtb input.dts`
Dump merged DT	`dtc -O dts -o out.dts out.dtb`
Avoid referencing overlay symbols	Only refer to symbols in base DT
Indicate overlays to kernel	Use `androidboot.dtbo_idx=3,5,...`
Validate with VTS	Check `ro.boot.dtbo_idx` and overlays

Would you like me to give you a full working example (main `.dts`, overlay `.dts`, simulated merged `.dts`, and commands)?

16. explain each code in simple words

Absolutely! Let’s break down and explain **each example code** related to **DTO (Device Tree Overlay)** syntax in **simple words**, so you understand **what each line does** and **why it's written that way**.

🔷 1. Overlay using old `fragment@x` + `__overlay__` (NOT recommended)

```dts
/ {
  fragment@0 {
    target = <&some_node>;
    __overlay__ {
      some_prop = "okay";
      ...
    };
  };
};
```

🧾 **Explanation**:

- **`fragment@0`**: This is an overlay section targeting one part of the main DT.
- **`target = <&some_node>;`**: Says "I want to modify the node called **`some_node`**".
- **`__overlay__`**: This block tells the compiler: "Add or change these properties in **`some_node`**".
- **`some_prop = "okay";`**: This adds/updates the property **`some_prop`** in **`some_node`**.

⚠️ This is the **older style**. Google recommends using the newer **reference syntax**.

🔷 2. Overlay using reference syntax (✅ recommended)

```dts
&some_node {
  some_prop = "okay";
  ...
};
```

🧾 **Explanation**:

- **`&some_node`**: This is a reference to a node defined in the main DT.
- **`some_prop = "okay";`**: This adds or updates the property **`some_prop`** inside that node.

✅ Cleaner. No **`fragment@x`**. Easier to read and write.

🔷 3. Use labels and references across overlay

Main DT:

```dts
/dts-v1/;

/ {
  my_node: node@0 {
    status = "disabled";

    my_child: child@0 {
      value = <0xffffffff>;
    };
  };
};
```

Overlay DT:

```dts
/dts-v1/;
/plugin/;

&my_node {
  status = "okay";
};

&my_child {
  value = <0x1>;
};
```

🧾 **Explanation**:

- `my_node:` and `my_child:` are **labels** added in main DT so you can reference them from overlay.
- `status = "disabled";` — means this device/node is initially turned off.
- Overlay says:
  - `&my_node { status = "okay"; };` — Turn it ON.
  - `&my_child { value = <0x1>; };` — Change value property.

🧠 Label + reference mechanism is the **key technique** in overlays.

🔷 4. Override existing property

Main DT:

```dts
/dts-v1/;

/ {
  compatible = "corp,foo";

  my_node: node@0 {
    status = "disabled";
  };
};
```

Overlay DT:

```dts
/dts-v1/;
/plugin/;

&my_node {
  status = "okay";
};
```

Merged Result:

```dts
/dts-v1/;

/ {
  compatible = "corp,foo";

  node@0 {
    status = "okay";
```

17. Figure 1. Use syntax /include/ to simulate DTO on the host.

Create a copy of the overlay .dts. In the copy, remove the first line header. Example:

```
/dts-v1/;
/plugin/;
```

Save the file as my_overlay_dt_wo_header.dts (or any filename you want).
Create a copy of the main .dts. In the copy, after the last line, append the include syntax for the file you created in step 1. For example:

```
/include/ "my_overlay_dt_wo_header.dts"
```

Save the file as my_main_dt_with_include.dts (or any filename you want).
Use dtc to compile my_main_dt_with_include.dts to get the merged DT, which should be the same result as DTO. For example:

```
dtc -@ -O dtb -o my_merged_dt.dtb my_main_dt_with_include.dts
```

Use dtc to dump my_merged_dt.dto.

```
dtc -O dts -o my_merged_dt.dts my_merged_dt.dtb
```

Verify DTO in Android 9
Android 9 requires a device tree blob overlay (DTBO) partition. To add nodes or make changes to the properties in the SoC DT, the bootloader must dynamically overlay a device specific DT over the SoC DT.

Indicate applied overlays
To enable the Vendor Test Suite (VTS) to assess the correctness of overlay app, vendors must add a new kernel command line parameter androidboot.dtbo_idx that indicates the overlays selected from the DTBO partition. In Android 12 using kernel version 5.10 or greater, this parameter passes through bootconfig. For example, the parameter androidboot.dtbo_idx=x,y,z reports x, y and z as the zero-based indices of the DTOs from the DTBO partition applied (in that order) by the bootloader to the base DT.

Overlays can apply to nodes from the main DT or add new nodes, but can't refer to a node added in a previous overlay. This restriction is necessary because the overlay app doesn't merge the overlay symbol table with the main DT symbol table (not merging avoids conflicts in symbol names and complication of dependencies between overlays).

Example: Invalid overlays
In this example, overlay_2.dts refers to node e , which was added by overlay_1.dts. After overlay_1 is applied to the main DT, if an attempt is made to apply overlay_2 to the resultant DT, the overlay app fails with an error that the symbol e isn't present in the symbol table for the base DT.

main.dts   overlay_1.dts   overlay_2.dts

[main.dts]

```
/dts-v1/;

/ {
  a: a {};
  b: b {};
  c: c {};
};
```

[overlay_1.dts]

```
/dts-v1/;
/plugin/;

&b { ref1 =  <&a>;
    e: e {
        prop = <0x0a>;
        phandle = <0x04>;
    };
};
```

[overlay_2.dts]

```
/dts-v1/;
/plugin/;
```

```
/* invalid! */
&e {
   prop = <0x0b>;
};
```

Example: Valid overlays
In this example, overlay_2.dts refers only to node b from the main DTS. When overlay_1 is applied to the base DT, then followed by the app of overlay_2, the value of property prop in node e (set by overlay_1.dts) is overwritten by the value set by overlay_2.dts.

main.dts   overlay_1.dts   overlay_2.dts

[final.dts]

```
/dts-v1/;

/ {
  a: a {};
  b: b {};
  c: c {};
};
```

[overlay_1.dts]

```
/dts-v1/;
/plugin/;


&b { ref1 =  <&a>;
     e {
          prop = <0x0c>;
      };
};
```

[overlay_2.dts]

```
/dts-v1/;
/plugin/;

/* valid */
&b { ref1 =  <&c>;
     e {
          prop = <0x0d>;
      };
};
```

Implement the DTBO partition
To implement the required DTBO partition, ensure the bootloader can do the following:

Identify the board it is running on and select the corresponding overlays to be applied.
Append the androidboot.dtbo_idx parameter to the kernel command line.
The parameter must indicate, the zero-based indices of the DTOs from the DTBO partition image it applied to the base DT (in the same order).
The indices must refer to the position of the overlay in the DTBO partition.
For details on the structure of the DTBO partition, refer to Device tree overlays.

Validate the DTBO partition
You can use VTS to verify the following:

Existence of the kernel command line parameter androidboot.dtbo_idx (by checking that Init has automatically set up the corresponding ro.boot.dtbo_idx system property).
Validity of the ro.boot.dtbo_idx system property (by checking that the property specifies at least one valid DTBO image index).
Validity of the DTBO partition (also verifies the overlays in the DTBO partition that are applied to the base DT).
Additional nodes or property changes in the resulting DT are presented to the Linux kernel.
For example, in the following overlays and final DT, adding androidboot.dtbo_idx=5,3 to the kernel command line passes validation but adding androidboot.dtbo_idx=3,5 to the kernel command line doesn't pass validation.

Overlay DT at index 3 Overlay DT at index 5

```
[overlay_1.dts]

/dts-v1/;
/plugin/;

&c { prop = <0xfe>; };

[overlay_2.dts]

/dts-v1/;
/plugin/;

&c { prop = <0xff>; };
Final DT

/dts-v1/;
/ {

	a {
		phandle = <0x1>;
	};

	b {
		phandle = <0x2>;
	};

	c {
		phandle = <0x3>;
		prop = <0xfe>;
	};

	__symbols__ {
		a = "/a";
		b = "/b";
		c = "/c";
	};
};
```

Here's a **simple explanation** of the entire section you posted — including how to simulate DTO on your development machine, why `androidboot.dtbo_idx` matters, and what the overlay examples are showing.

✅ What's Being Done?

You're learning:

- How to **test** Device Tree Overlays (DTOs) **before flashing to a real device**.
- How to apply **overlays safely**, and
- What can go **wrong or right** in DTO applications (valid/invalid overlay examples).

🔷 1. Simulating DTO on Host using `/include/` (instead of real overlay tool)

▯ Purpose:

To simulate the result of overlaying an overlay DT (`.dts`) on a main DT (`.dts`) **on your PC** — without a bootloader.

▯ Steps:

🧩 Step 1: Create a copy of overlay without header

Original:

```dts
/dts-v1/;
/plugin/;

&some_node {
    property = "value";
};
```

Modified version (saved as **`my_overlay_dt_wo_header.dts`**):

```dts
&some_node {
    property = "value";
};
```

> ✂️ Remove **`/dts-v1/;`** and **`/plugin/;`** so it can be **included**.

🧩 Step 2: Create new main DTS file and include overlay

```dts
/dts-v1/;

/ {
    ...
};

/include/ "my_overlay_dt_wo_header.dts"
```

> 📄 Save this file as **`my_main_dt_with_include.dts`**.

🛠️ Step 3: Compile using DTC

```bash
dtc -@ -O dtb -o my_merged_dt.dtb my_main_dt_with_include.dts
```

- **`-@`**: Adds label symbols.
- **`-O dtb`**: Output is a **`.dtb`** file.

🧾 Step 4: Convert back to readable `.dts`

```bash
dtc -O dts -o my_merged_dt.dts my_merged_dt.dtb
```

Now **`my_merged_dt.dts`** is the result **you would get** if your bootloader merged the overlay on the device.

🔷 2. `androidboot.dtbo_idx` (Important in Android 9+)

- This is a **kernel command line parameter** added by the bootloader.
- It tells the kernel: **Which DTBO overlay(s)** were applied.

Example:

```bash
androidboot.dtbo_idx=2,4
```

→ Means overlays at **index 2 and 4** from the DTBO partition were applied.

📋 Why It's Needed?

- For **VTS (Vendor Test Suite)** validation.
- Helps Android understand what overlays are active.

🔷 3. ❌ Invalid Overlay Example

❗ Problem:

Overlay_2 is trying to modify node `e`, but that node was created in overlay_1 — not in the base (main) DT.

```dts
// main.dts
/ {
  a: a {};
  b: b {};
  c: c {};
};
```

```dts
// overlay_1.dts
&b {
    ref1 = <&a>;
    e: e {
        prop = <0x0a>;
    };
};
```

```dts
// overlay_2.dts
// ❌ INVALID — refers to `e` which doesn't exist in the base
&e {
    prop = <0x0b>;
};
```

Overlay tools don't **merge overlay symbol tables**, so `e` is unknown in overlay_2. ❌

🔷 4. ✅ Valid Overlay Example

```dts
// main.dts
/ {
  a: a {};
  b: b {};
  c: c {};
};
```

```dts
// overlay_1.dts
&b {
    ref1 = <&a>;
    e {
        prop = <0x0c>;
    };
};
```

```dts
// overlay_2.dts
// ✅ VALID — modifies `e` again, but inside `&b`, not directly referencing `&e`
&b {
    ref1 = <&c>;
    e {
        prop = <0x0d>;
    };
};
```

Overlay_2 doesn't rely on symbol `e` — just modifies structure under `b`. This is allowed. ✅

🔷 5. DTBO Partition Requirements

To **implement the DTBO partition correctly**, the bootloader must:

1. **Know which board it's on**
2. Choose correct overlays
3. Append this to kernel command line:

   ```bash
   androidboot.dtbo_idx=3,5
   ```

Indices refer to position in DTBO image.

🔷 6. VTS Validation Checks

Vendor Test Suite will:

✅ Check for:

- `ro.boot.dtbo_idx` system property
- If indices are valid
- That the overlays applied properly modify the DT

🔷 7. Example Validation Failure

✅ This works:

```bash
androidboot.dtbo_idx=5,3
```

Overlay 5 overrides the changes from overlay 3 — **last write wins**.

❌ This fails:

```bash
androidboot.dtbo_idx=3,5
```

Same overlays, but applied in wrong order. Causes VTS to fail.

🔚 Summary

Concept	Explained Simply
/include/	Simulate overlay by merging DTS files offline
-@	DTC flag to add symbol info
androidboot.dtbo_idx	Tells kernel which overlays were applied
Valid overlay	Doesn’t depend on nodes defined by other overlays
DTBO partition	Holds overlays; must be verifiable and ordered
VTS checks	Ensures overlays work and match declared indices

Would you like an illustrated version of this flow or code-to-diagram explanation?

18. confusing

No problem — let’s break it down in **very simple terms**, one step at a time. We'll cover:

💡 What Is DTO (Device Tree Overlay)?

Think of a **Device Tree (DT)** like a blueprint for your hardware.

- The **main DT** (from SoC vendor) has the default design (e.g., CPU, RAM).
- The **overlay DT** (from OEM/ODM) adds/changes things (e.g., touchscreen, sensors).

📦 **DTO** = Combine the main DT + overlay DT at runtime.

🧪 Why Simulate DTO?

Instead of flashing to a real device and testing there, you can **simulate DTO on your PC** using:

✅ `/include/` to test merge
✅ `dtc` tool to compile and dump result

🧩 Step-by-Step: Simulating DTO

Step 1: Start with two `.dts` files

Main DTS (`my_main_dt.dts`)

```dts
/dts-v1/;

/ {
  my_node: node@0 {
    status = "disabled";
  };
};
```

Overlay DTS (`my_overlay_dt.dts`)

```dts
/dts-v1/;
/plugin/;

&my_node {
  status = "okay";
};
```

> 🔍 This overlay will change `status = "disabled"` to `status = "okay"`.

Step 2: Create "include" version of overlay

Make a copy of `my_overlay_dt.dts`, remove the first two lines:

```dts
&my_node {
  status = "okay";
};
```

Save this as `my_overlay_dt_wo_header.dts`.

Step 3: Include overlay in main DTS

Create new main DTS file:

```dts
/dts-v1/;

/ {
  my_node: node@0 {
    status = "disabled";
  };
};

/include/ "my_overlay_dt_wo_header.dts"
```

Save this as **`my_main_dt_with_include.dts`**.

Step 4: Compile and Dump

Compile to .dtb

```bash
dtc -@ -O dtb -o my_merged_dt.dtb my_main_dt_with_include.dts
```

Dump back to readable format

```bash
dtc -O dts -o my_merged_dt.dts my_merged_dt.dtb
```

✅ You'll see:

```dts
my_node: node@0 {
  status = "okay";
};
```

That means: **overlay applied successfully**.

❌ Invalid Overlay Example (Simple)

Let's say:

- Overlay 1 creates a node **`e`**
- Overlay 2 tries to **reference** that node **`e`**

That fails ❌ — because the base DT doesn't know about **`e`**.

✅ Valid Overlay Example

Overlay 1:

```dts
&b {
  e {
    prop = <0x0c>;
  };
};
```

Overlay 2:

```dts
&b {
  e {
    prop = <0x0d>;
  };
};
```

Valid ✅ — both modify node under `b`, not depending on each other.

🧠 What is `androidboot.dtbo_idx`?

When bootloader applies overlays, it must tell the kernel which ones:

```
bash

androidboot.dtbo_idx=3,5
```

Means: "I applied overlay 3 first, then overlay 5."

If the order is wrong (e.g., 5 before 3), kernel may break.

🧪 VTS Validation

Vendor Test Suite checks:

- That `ro.boot.dtbo_idx` is set
- That the listed overlays are valid
- That the final device tree works

🎯 Summary

Concept	Simple Meaning
DTO	Add/modify hardware config at boot
Main DTS	Base hardware layout
Overlay DTS	Changes or adds to the base layout
/plugin/	Tells compiler it's an overlay
/include/	Way to merge DTs offline for testing
androidboot.dtbo_idx	List of overlay indices applied at boot
dtc	Compiler for DTS <-> DTB
Valid overlays	Don't depend on other overlays

Would you like a **real code demo** or **visual diagram** to better understand this?

19. Use multiple DTs

bookmark_border
Many SoC vendors and ODMs support the use of multiple device trees (DTs) on a device, enabling one image to power multiple SKUs or configurations. In such cases, the bootloader identifies the hardware and loads the corresponding DT at runtime:

Figure 1. Multiple device tree overlays (DTOs) in bootloader.

Note: Using multiple DTs isn't mandatory.

Set up
To add support for multiple DTs to the DTO model, set up a list of main DTs and another list of overlay DTs.

Figure 2. Runtime DTO implementation for multiple DTs.

The bootloader should be able to:

Read the SoC ID and select the corresponding main DT.
Read the board ID and select the set of overlay DTs accordingly.
Only one main DT should be selected for use at runtime. Multiple overlay DTs may be selected but they must be compatible with the chosen main DT. Using multiple overlays can help avoid storing one overlay per board within the DTBO partition and enable the bootloader to determine the subset of required overlays based on the board ID (or possibly by probing the peripherals). For example, Board A may need the devices added by the overlays 1, 3, and 5 while Board B may need the devices added by the overlays 1, 4, and 5.

Partition
To partition, determine a bootloader runtime-accessible and trusted location in flash memory to store the DTBs and DTBOs (bootloader must be able to locate these files in the matching process). Keep in mind that DTBs and DTBOs cann't exist in the same partition. If your DTBs/DTBOs are in the dtb/dtbo partition, use the table structure and header format detailed in DTB and DTBO partition format.

Run in bootloader
To run:

Identify the SoC and load the corresponding .dtb from storage into memory.
Identify the board and load the corresponding .dtbo from storage into memory.
Overlay the .dtb with the .dtbo to be a merged DT.
Start kernel given the memory address of the merged DT.

Great! Let's **explain the concept of using multiple Device Trees (DTs)** — in **simple words** — especially how it's handled at **boot time** with the help of the bootloader.

🧠 What is this about?

When a single firmware image needs to support **multiple hardware variants** (called SKUs or boards), we **don't want to build a different image for each**. Instead:

✅ Use **multiple Device Tree files** (DTs and overlays)
✅ Let the **bootloader decide** which DT and overlays to use

🔧 Problem This Solves

Imagine you have:

- One SoC (e.g., Snapdragon 8 Gen 2)
- But 3 different phones built with it:
  - Phone A (has NFC + 5G)

  - Phone B (no NFC, just 4G)
  - Phone C (has extra camera)

Each phone uses the **same kernel and OS**, but needs **different device configurations**.

✅ Solution: Use Multiple DTs

- **Main DTB (.dtb)** = Describes the SoC + base hardware
- **Overlay DTBO (.dtbo)** = Describes device-specific things (extra sensors, cameras, etc.)

The **bootloader**:

1. Detects which SoC and board is running
2. Loads:
   - The right `.dtb` for the SoC
   - One or more `.dtbo` overlays for the board
3. Merges `.dtb + .dtbo`
4. Boots the kernel with the final merged DT

🧱 How It's Structured

🗂️ Partitioning

File Type	Description	Partition (Example)
`.dtb`	Main DT from vendor	`dtb_a` / `dtb_b`
`.dtbo`	Overlay DT from ODM/OEM	`dtbo_a` / `dtbo_b`

> 🔒 These are in **trusted, bootloader-accessible partitions**

🏃 Bootloader Runtime Flow

1. 📍 **Identify SoC** → Load `main.dtb`
2. 📍 **Identify Board ID** → Pick matching overlays:
   - e.g., Board A: apply overlays 1, 3, 5
   - e.g., Board B: apply overlays 1, 4, 5
3. 🧩 Merge `.dtb + .dtbo`
4. 🚀 Start the kernel with the merged DT in memory

🔄 Why Use Multiple Overlays?

Let's say:

- Overlay 1 = touchscreen
- Overlay 3 = NFC
- Overlay 4 = fingerprint sensor
- Overlay 5 = special GPIO config

By **mixing and matching overlays**, you avoid:
❌ One big overlay per board
✅ Reuse overlays across boards (saves space and effort)

🧪 Example Bootloader Logic

```pseudo
if SoC_ID == QCOM_8_GEN_2:
    load main_dtb = dtb[2]

if Board_ID == "board_A":
    load overlays = [overlay1, overlay3, overlay5]
elif Board_ID == "board_B":
    load overlays = [overlay1, overlay4, overlay5]

merged_dt = apply_overlays(main_dtb, overlays)
boot_kernel(merged_dt)
```

📂 Partition Format

If using `dtb` and `dtbo` partitions:

- Use the **DT table format** (with headers and entries)
- Each entry in `.dtbo` partition can be indexed
- You reference overlays by **index** (like 0, 1, 2...)

📌 Summary

Term	Meaning
DTB	Main device tree (SoC-level)
DTBO	Overlay device tree (board-specific config)
Bootloader	Picks and merges correct DTs at runtime
Multiple DTs	Supports many boards/SKUs with one image
Partitions	Store DTB and DTBOs separately

Would you like a **visual diagram or code snippet** to help better understand this setup?

20. ⤷
📂 Partition Format ------------------- If using `dtb` and `dtbo` partitions: * Use the **DT table format** (with headers and entries) * Each entry in `.dtbo` partition can be indexed * You reference overlays by **index** (like 0, 1, 2...)
this is confusing

You're right — that part is tricky. Let's simplify the **partition format for DTB and DTBO** using **real-world comparisons** and **easy words**.

📦 What Are DTB and DTBO Partitions?

In Android:

- **`dtb` partition** holds the **main device trees** (`*.dtb`)
- **`dtbo` partition** holds the **overlay device trees** (`*.dtbo`)

Each of these partitions **contains multiple entries**, like a folder full of files.

📚 Think of It Like a Book 📖

Imagine the **DTBO partition** is like a book:

- It has a **table of contents** at the front (called a **DT Table Header**)
- Each "chapter" is an overlay (called a **DTBO entry**)
- You can say: *"Load overlay number 3"* → and the bootloader jumps to entry 3

This is what we mean by **index-based access**.

🧠 Why Indexing Is Used?

Instead of:

- Naming overlays like `touch_overlay`, `camera_overlay`, etc.

The bootloader says:

```bash
androidboot.dtbo_idx=1,3,5
```

This means:

> Load overlay at index 1, then 3, then 5 from the DTBO partition.

The **bootloader knows** which overlays are at which positions.

💾 What Is in the DTBO Partition?

The **DTBO partition** contains:

1. A **header** (called `dt_table_header`)
2. A list of **entries** (called `dt_table_entry[]`)
3. The actual binary **DTBOs**

Every `.dtbo` file is described by:

- Offset in the partition
- Size

- Version info
- Hardware compatibility info

This allows the bootloader to:

- Read all entries
- Pick the ones matching your board
- Load them in order

🔧 Who Decides Which Indexes to Load?

The **bootloader** logic says:

```text
If Board_ID == "A":
    apply overlays at index 1, 3, 5

If Board_ID == "B":
    apply overlays at index 0, 2
```

So when booting, it passes:

```bash
androidboot.dtbo_idx=1,3,5
```

This tells the kernel **which overlays were applied**, so validation tools (like VTS) can verify it.

✅ Summary

Concept	Simple Meaning
`dtb` partition	Stores main SoC-level device trees
`dtbo` partition	Stores overlay device trees
DT Table Header	Like a table of contents
DTBO index	Like a chapter number in the DTBO "book"
`androidboot.dtbo_idx`	Tells kernel which overlays bootloader applied
Bootloader job	Picks DTB and DTBO entries based on hardware

Would you like a diagram of how the DTBO table works visually?

21. ok

Generated image

22. explain this image

This image illustrates how **Device Tree Overlays (DTOs)** are used at runtime in Android's bootloader to build a final **Merged Device Tree (DT)** before the kernel starts.

🔷 Left Side: Input Files

1. Main DT (`my_main_dt.dts`)

- This is the base Device Tree Source file.
- Describes core hardware provided by the **SoC vendor**.
- Starts with `/dts-v1/;` and contains nodes and properties for SoC-level components.

2. Overlay DT (`my_overlay_dt.dts`)

- This is an overlay Device Tree Source used to **customize or add** board-specific details (e.g., extra devices).
- Has `/plugin/;` tag to signal it's an overlay.
- Adds/modifies nodes in the main DT.

🔷 Middle: Simulation on Host

Instead of booting the image to test overlays, you simulate the merge on your host machine using:

🟨 Modified Files:

- `my_overlay_dt_wo_header.dts`: A copy of the overlay with the first line (`/dts-v1;/plugin;`) **removed**, to allow including.
- `my_main_dt_with_include.dts`: A copy of the main DT that includes the overlay with `/include/ "my_overlay_dt_wo_header.dts"` at the end.

This simulates how bootloader will merge overlay with the base DT.

🔷 Bottom Left: Compile

Use `dtc` (Device Tree Compiler) to:

1. Compile the simulated DTS file:

   ```bash
   dtc -@ -O dtb -o my_merged_dt.dtb my_main_dt_with_include.dts
   ```

   - `-@`: Adds symbol info required for overlays.
   - `-O dtb`: Output in binary format.
2. Convert compiled `.dtb` back to readable `.dts` to inspect:

   ```bash
   dtc -O dts -o my_merged_dt.dts my_merged_dt.dtb
   ```

🔷 Right Side: Merged DT

- Final result after merging the overlay into the main DT.
- This DT is what the **bootloader passes to the kernel**.
- Contains both SoC and board-specific configurations.

✅ Purpose

This method helps **verify correctness of overlays** without flashing the image. It:

- Prevents overlay errors at runtime.
- Makes debugging easier.
- Ensures compatibility between base DT and overlays before boot.

Let me know if you'd like me to break down an example `.dts` and `.dtbo` file line by line.

23. DTB and DTBO partitions

bookmark_border
If your device tree blob (DTB) or device tree blob for overlay (DTBO) is in a unique partition, for example, the dtb and dtbo partition, use the following table structure and header format:

Data structures
The dt_table_header is only for the dtb/dtbo partition; you CAN'T append this format after the end of image.gz. If you have a single DTB or DTBO, you must still use this format (and the dt_entry_count in dt_table_header is 1).

```
#define DT_TABLE_MAGIC 0xd7b7ab1e

struct dt_table_header {
  uint32_t magic;             // DT_TABLE_MAGIC
  uint32_t total_size;        // includes dt_table_header + all dt_table_entry
                              // and all dtb/dtbo
  uint32_t header_size;       // sizeof(dt_table_header)

  uint32_t dt_entry_size;     // sizeof(dt_table_entry)
  uint32_t dt_entry_count;    // number of dt_table_entry
  uint32_t dt_entries_offset; // offset to the first dt_table_entry
                              // from head of dt_table_header

  uint32_t page_size;         // flash page size we assume
  uint32_t version;       // DTBO image version, the current version is 0.
                          // The version is incremented when the
                          // dt_table_header struct is updated.
};

struct dt_table_entry {
  uint32_t dt_size;
  uint32_t dt_offset;         // offset from head of dt_table_header

  uint32_t id;                // optional, must be zero if unused
  uint32_t rev;               // optional, must be zero if unused
  uint32_t custom[4];         // optional, must be zero if unused
};
```

To read all dt_table_entry, use the dt_entry_size, dt_entry_count, and dt_entries_offset. Example:

```
my_read(entries_buf,
        header_addr + header->dt_entries_offset,
        header->dt_entry_size * header->dt_entry_count);
```

The id, rev, custom in dt_table_entry are optional hardware identifications of the device tree the bootloader can use to efficiently identify the DTB or DTBO to load. If the bootloader requires additional information, put it in the DTB or DTBO where bootloader can read it by parsing DTB or DTBO (see the sample code below).

Sample code
The following sample code checks the hardware identification in bootloader.

The check_dtbo() function checks the hardware identification. It first checks the data in struct dt_table_entry (id, rev, etc.). If this data isn't enough, it loads dtb data into memory and checks the value in dtb.
The values of my_hw_information and soc_id properties are parsed in the root node (example in my_dtbo_1.dts).

```
[my_dtbo_1.dts]
/dts-v1/;
/plugin/;

/ {
  /* As DTS design, these properties only for loader, won't overlay */
  compatible = "board_manufacturer,board_model";

  /* These properties are examples */
  board_id = <0x00010000>;
  board_rev = <0x00010001>;
  another_hw_information = "some_data";
  soc_id = <0x68000000>;
  ...
};

&device@0 {
  value = <0x1>;
  status = "okay";
};
```

```
[my_bootloader.c]
int check_dtbo(const dt_table_entry *entry, uint32_t header_addr) {
  ...
  if (entry->id != ... || entry->rev != ...) {
    ...
  }
  ...
  void * fdt_buf = my_load_dtb(header_addr + entry->dt_offset, entry->dt_size);
  int root_node_off = fdt_path_offset(fdt_buf, "/");
  ...
  const char *my_hw_information =
    (const char *)fdt_getprop(fdt_buf, root_node_off, "my_hw_information", NULL);
  if (my_hw_information != NULL && strcmp(my_hw_information, ...) != 0) {
    ...
  }
  const fdt32_t *soc_id = fdt_getprop(fdt_buf, root_node_off, "soc_id", NULL);
  if (soc_id != NULL && *soc_id != ...) {
    ...
  }
  ...
}
```

mkdtimg

mkdtimg is a tool for creating dtb/dtbo images (source code at system/libufdt in AOSP). mkdtimg supports several commands, including create, cfg_create, and dump.

create

Use the create command to create a dtb/dtbo image:

```
mkdtimg create <image_filename> (<global-option>...) \
    <ftb1_filename> (<entry1_option>...) \
    <ftb2_filename> (<entry2_option>...) \
    ...
```

ftbX_filename generates a dt_table_entry in the image. entryX_options are the values to assign to dt_table_entry. These values can be any of the following:

```
--id=<number|path>
--rev=<number|path>
--custom0=<number|path>
--custom1=<number|path>
--custom2=<number|path>
--custom3=<number|path>
```

Number values can be a 32-bit digit (such as 68000) or a hex number (such as 0x6800). Alternatively, you can specify a path using the format:

```
<full_node_path>:<property_name>
```

For example, /board/:id. mkdtimg reads the value from the path in the DTB or DTBO file and assigns the value (32-bit) to a relative property in dt_table_entry. Alternatively, you can give a global_option as a default option for all entries. The default value of page_size in dt_table_header is 2048; use global_option --page_size=<number> to assign a different value.

Example:

```
[board1.dts]
/dts-v1/;
/plugin/;

/ {
  compatible = "board_manufacturer,board_model";
  board_id = <0x00010000>;
  board_rev = <0x00010001>;
  another_hw_information = "some_data";
  ...
};

&device@0 {
  value = <0x1>;
  status = "okay";
};
```

```
mkdtimg create dtbo.img --id=/:board_id --custom0=0xabc \
  board1.dtbo \
  board2.dtbo --id=0x6800 \
  board3.dtbo --id=0x6801 --custom0=0x123
```

First dt_table_entry (board1.dtbo) id is 0x00010000 and custom[0] is 0x00000abc.
Second id is 0x00006800 and custom[0] is 0x00000abc.
Third id is 0x00006801 and custom[0] is 0x00000123.
All others use the default value (0).
cfg_create
The cfg_create command creates an image with a config file in the following format:

```
# global options
  <global_option>
  ...
# entries
<ftb1_filename>     # comment
  <entry1_option>   # comment
  ...
<ftb2_filename>
  <entry2_option>
  ...
...
```

Options global_option and entryX_option must start with one or more space characters (these options are the same as create options, without the -- prefix). Empty lines or lines beginning with # are ignored.

Example:

```
[dtboimg.cfg]
# global options
  id=/:board_id
  rev=/:board_rev
  custom0=0xabc

board1.dtbo

board2.dtbo
  id=0x6800       # override the value of id in global options

board2.dtbo
  id=0x6801       # override the value of id in global options
  custom0=0x123   # override the value of custom0 in global options
```

mkdtimg cfg_create dtbo.img dtboimg.cfg
mkdtimg doesn't handle alignment for .dtb/.dtbo files but rather appends them to the image. When you use dtc to compile .dts to .dtb/.dtbo, you must add option -a. For example, adding the option -a 4 adds padding so the size of .dtb/.dtbo aligns to 4 bytes.

Several DT table entries can share a .dtb/.dtbo. If you use the same filename for different entries, it stores only one content in the image with same dt_offset and dt_size. This is useful when using different hardware with identical DTs.

dump
For dtb/dtbo images, use the dump command to print the information in the image. Example:

```
mkdtimg dump dtbo.img
dt_table_header:
               magic = d7b7ab1e
          total_size = 1300
         header_size = 32
       dt_entry_size = 32
      dt_entry_count = 3
   dt_entries_offset = 32
           page_size = 2048
             version = 0
dt_table_entry[0]:
             dt_size = 380
           dt_offset = 128
                  id = 00010000
                 rev = 00010001
           custom[0] = 00000abc
           custom[1] = 00000000
           custom[2] = 00000000
           custom[3] = 00000000
           (FDT)size = 380
     (FDT)compatible = board_manufacturer,board_model
...
```

Let's break this **DTB/DTBO partition** concept into simple steps:

🔧 What Is This About?

Modern Android devices often use **Device Trees (DTs)** to describe hardware. These DTs come in two types:

- **DTB** = Main **Device Tree Blob**
- **DTBO** = **Overlay** Device Tree (adds/changes stuff in main DTB)

To manage **multiple device variants (SKUs)**, Android stores these DTBs and DTBOs in **dedicated partitions**:

- `dtb` partition → holds multiple **main DTBs**
- `dtbo` partition → holds multiple **overlays (DTBOs)**

The format used inside these partitions is a **structured table**, described below 👇

🧱 1. DT Table Format (in partition)

This is how a partition like `dtbo.img` is structured:

✅ `dt_table_header`

This is like a table of contents at the beginning:

```c
struct dt_table_header {
  uint32_t magic;             // Always 0xd7b7ab1e
  uint32_t total_size;        // Total size of image
  uint32_t header_size;       // Size of this struct
  uint32_t dt_entry_size;     // Size of each entry
  uint32_t dt_entry_count;    // Number of entries
  uint32_t dt_entries_offset; // Where the list of entries starts
  uint32_t page_size;         // Flash alignment (e.g. 2048)
  uint32_t version;           // Current = 0
};
```

✅ `dt_table_entry`

Each entry in the list tells where one `.dtb` or `.dtbo` is located:

```c
struct dt_table_entry {
  uint32_t dt_size;     // Size of .dtb/.dtbo blob
  uint32_t dt_offset;   // Where it starts in file

  uint32_t id;          // Board ID (optional)
  uint32_t rev;         // Board revision (optional)
  uint32_t custom[4];   // Optional extra info
};
```

📦 2. Example `.dts` with metadata (used in overlay)

Here's what a typical overlay DTS looks like:

```dts
/dts-v1/;
/plugin/;

/ {
  compatible = "board_manufacturer,board_model";
  board_id = <0x00010000>;
  board_rev = <0x00010001>;
  another_hw_information = "some_data";
  soc_id = <0x68000000>;
};

&device@0 {
  value = <0x1>;
  status = "okay";
};
```

- The **metadata fields** (`board_id`, `soc_id`, etc.) are not applied to the kernel but help the bootloader decide which overlay to apply.

⚙️ 3. Bootloader Chooses the Right DTBO

Sample C code (`check_dtbo`) checks each DTBO entry:

```c
// Check ID and revision
if (entry->id != expected_id || entry->rev != expected_rev)
    skip_this_entry;

// Load DTBO from dt_offset
void *dtb = my_load_dtb(header + entry->dt_offset, entry->dt_size);

// Parse property like soc_id inside the DTB itself
int root = fdt_path_offset(dtb, "/");
const fdt32_t *soc_id = fdt_getprop(dtb, root, "soc_id", NULL);
```

🛠️ 4. Creating DTBO Images with `mkdtimg`

The tool `mkdtimg` is used to pack `.dtbo` files into a single `dtbo.img`.

🧪 Option 1: CLI (direct)

```bash
mkdtimg create dtbo.img \
  board1.dtbo --id=/:board_id --custom0=0xabc \
  board2.dtbo --id=0x6800 \
  board3.dtbo --id=0x6801 --custom0=0x123
```

💡 **`--id=/:board_id`** means:

- Look into **`board1.dtbo`**, find property **`board_id`** in root `/`, use that value as ID.

🧪 Option 2: Config File (`cfg_create`)

Instead of passing options in command-line, use a **`.cfg`** file:

```
cfg

# dtboimg.cfg

  id=/:board_id
  rev=/:board_rev
  custom0=0xabc

board1.dtbo

board2.dtbo
  id=0x6800

board3.dtbo
  id=0x6801
  custom0=0x123
```

Build with:

```
bash

mkdtimg cfg_create dtbo.img dtboimg.cfg
```

🧐 5. Inspecting with `mkdtimg dump`

To debug and inspect the final **`.img`**:

```
bash

mkdtimg dump dtbo.img
```

Sample output:

```
yaml

dt_table_header:
  magic = d7b7ab1e
  total_size = 1300
  dt_entry_count = 3
  page_size = 2048

dt_table_entry[0]:
  dt_size = 380
  id = 00010000
  rev = 00010001
  custom[0] = 00000abc
  compatible = board_manufacturer,board_model
```

✅ Summary

Term	Meaning
DTB	Main hardware description from SoC vendor
DTBO	Overlay that modifies or adds new board-specific config
dtbo.img	A collection of DTBOs packaged using a table format
Bootloader	Selects and applies DTBO based on hardware info
mkdtimg	Tool to build, configure, and inspect DTBO images

If you want, I can also show you a **real-world AOSP folder structure**, or guide you through **using mkdtimg step-by-step**.

24. Optimize DTOs

bookmark_border
This page discusses optimizations you can make to your device tree overlay (DTO) implementation, describes restrictions against overlaying the root node, and details how to configure compressed overlays in the DTBO image. It also provides sample implementation instructions and code.

Kernel command line
The original kernel command line in device tree (DT) is located in the chosen/bootargs node. The bootloader must concatenate this location with other sources of kernel command line:

```
/dts-v1/;

/ {
  chosen: chosen {
    bootargs = "...";
  };
};
```

DTO cannot concatenate values from main DT and overlay DT, so you must put the kernel command line of the main DT in chosen/bootargs and the kernel command line of the overlay DT in chosen/bootargs_ext. Bootloader can then concatenate these locations and pass the result to the kernel.

main.dts   overlay.dts

```
/dts-v1/;

/ {
  chosen: chosen {
    bootargs = "...";
  };
};

/dts-v1/;
/plugin/;

&chosen {
  bootargs_ext = "...";
};
```

libufdt
While the latest libfdt supports DTO, is it recommended to use libufdt to implement DTO (AOSP source at platform/system/libufdt). libufdt builds a real tree structure (un-flattened device tree, or ufdt) from the flattened device tree (FDT), so it can improve the merging of two .dtb files from O(N2) to O(N), where N is the number of nodes in the tree.

Performance testing
In Google's internal testing, using libufdt on 2405 .dtb and 283 .dtbo DT nodes results in file sizes of 70,618 and 8,566 bytes after compilation. Compared with a DTO implementation ported from FreeBSD (124 ms runtime), libufdt DTO runtime is 10 ms.

Performance testing for Pixel devices compared libufdt and libfdt. The number of base nodes effect is similar, but includes the following differences:

500 overlay (append or override) operations have 6x to 8x time difference
1000 overlay (append or override) operations have 8x to 10x time difference
Example with appending count set to X:

libufdt is developed with some libfdt APIs and data structures. When using libufdt, you must include and link libfdt (however, in your code you can use the libfdt API to operate DTB or DTBO).

libufdt DTO API
The main API to DTO in libufdt is as follows:

```
struct fdt_header *ufdt_apply_overlay(
        struct fdt_header *main_fdt_header,
        size_t main_fdt_size,
        void *overlay_fdt,
        size_t overlay_size);
```

The parameter main_fdt_header is the main DT and overlay_fdt is the buffer containing the contents of a .dtbo file. The return value is a new buffer containing the merged DT (or null in case of error). The merged DT is formatted in FDT, which you can pass to the kernel when starting the kernel.

The new buffer from the return value is created by dto_malloc(), which you should implement when porting libufdt into bootloader. For reference implementations, refer to sysdeps/libufdt_sysdeps_*.c.

Root node restrictions

You cannot overlay a new node or property into the root node of main DT because overlay operations rely on labels. Because the main DT must define a label and the overlay DT assigns the nodes to be overlaid with labels, you cannot give a label for the root node (and therefore cannot overlay the root node).

SoC vendors must define the overlaying ability of main DT; ODM/OEMs can only append or override nodes with labels defined by the SoC vendor. As a workaround, you can define an odm node under the root node in base DT, enabling all ODM nodes in overlay DT to add new nodes. Alternatively, you could put all SoC-related nodes in the base DT into an soc node under root node as described below:

main.dts   overlay.dts

```
/dts-v1/;

/ {
    compatible = "corp,bar";
    ...

    chosen: chosen {
        bootargs = "...";
    };

    /* nodes for all soc nodes */
    soc {
        ...
        soc_device@0: soc_device@0 {
            compatible = "corp,bar";
            ...
        };
        ...
    };

    odm: odm {
        /* reserved for overlay by odm */
    };
};

/dts-v1/;
/plugin/;

/ {
};

&chosen {
    bootargs_ex = "...";
};

&odm {
    odm_device@0 {
        ...
    };
    ...
};
```

Use compressed overlays

Android 9 adds support for using compressed overlays in the DTBO image when using version 1 of the DT table header. When using DTBO header v1, the four least significant bits of the flags field in dt_table_entry indicate the compression format of the DT entry.

```
struct dt_table_entry_v1 {
  uint32_t dt_size;
  uint32_t dt_offset;  /* offset from head of dt_table_header */
  uint32_t id;         /* optional, must be zero if unused */
  uint32_t rev;        /* optional, must be zero if unused */
  uint32_t flags;      /* For version 1 of dt_table_header, the 4 least significant bits
                        of 'flags' are used to indicate the compression
                        format of the DT entry as per the enum 'dt_compression_info' */
  uint32_t custom[3];  /* optional, must be zero if unused */
};
```

Currently, zlib and gzip compressions are supported.

```
enum dt_compression_info {
    NO_COMPRESSION,
    ZLIB_COMPRESSION,
    GZIP_COMPRESSION
};
```

Android 9 adds support for testing compressed overlays to the VtsFirmwareDtboVerification test to help you verify the correctness of overlay app.

Sample DTO implementation
The following instructions walk you through a sample implementation of DTO with libufdt (sample code below).

Sample DTO instructions
Include libraries. To use libufdt, include libfdt for data structures and APIs:

```
#include <libfdt.h>
#include <ufdt_overlay.h>
```

Load main DT and overlay DT. Load .dtb and .dtbo from storage into memory (exact steps depend on your design). At this point, you should have the buffer and size of .dtb/.dtbo:

```
main_size = my_load_main_dtb(main_buf, main_buf_size)

overlay_size = my_load_overlay_dtb(overlay_buf, overlay_buf_size);
```

Overlay the DTs:
Use ufdt_install_blob() to get the FDT header for main DT:

```
main_fdt_header = ufdt_install_blob(main_buf, main_size);
main_fdt_size = main_size;
```

Call ufdt_apply_overlay() to DTO to get a merged DT in FDT format:

```
merged_fdt = ufdt_apply_overlay(main_fdt_header, main_fdt_size,
                                overlay_buf, overlay_size);
```

Use merged_fdt to get the size of dtc_totalsize():

```
merged_fdt_size = dtc_totalsize(merged_fdt);
```

Pass the merged DT to start the kernel:

```
my_kernel_entry(0, machine_type, merged_fdt);
```

Sample DTO code

```
#include <libfdt.h>
#include <ufdt_overlay.h>

…

{
  struct fdt_header *main_fdt_header;
  struct fdt_header *merged_fdt;

  /* load main dtb into memory and get the size */
  main_size = my_load_main_dtb(main_buf, main_buf_size);
  /* load overlay dtb into memory and get the size */
  overlay_size = my_load_overlay_dtb(overlay_buf, overlay_buf_size);

  /* overlay */
  main_fdt_header = ufdt_install_blob(main_buf, main_size);
  main_fdt_size = main_size;
  merged_fdt = ufdt_apply_overlay(main_fdt_header, main_fdt_size,
                              overlay_buf, overlay_size);
  merged_fdt_size = dtc_totalsize(merged_fdt);

  /* pass to kernel */
  my_kernel_entry(0, machine_type, merged_fdt);
}
```

Here's a **simple breakdown** of the whole **DTO Optimization** page you shared — organized step-by-step for clarity.

✅ Goal of This Page

It teaches you **how to optimize DTO (Device Tree Overlay)** in Android and explains:

- Kernel bootargs handling
- Faster overlaying with `libufdt`
- Root node overlay limitations
- How to use compressed DTBOs
- Sample implementation of DTO using `libufdt`

🧩 1. Handling Kernel Bootargs Properly

In Device Tree (DT), boot arguments are passed through:

```dts
/ {
  chosen {
    bootargs = "...";
  };
};
```

But in overlays, the bootargs need to go in a different property:

```dts
&chosen {
  bootargs_ext = "...";
};
```

🔧 Why?

Because **DTO can't merge or concatenate bootargs** across overlays.

✅ So, bootloader must:

- Read `bootargs` from base DT
- Read `bootargs_ext` from overlay DT
- Concatenate them and pass to the kernel

🚀 2. Use libufdt for Fast Overlaying

🛠️ What's libufdt?

- It's a library developed by Google (in AOSP)

- Converts .dtb into a **tree** (UFDT) so merging is **faster**
- Located in: `platform/system/libufdt`

🆚 Performance

Library	Merge Time (1000 nodes)
libfdt	~124 ms (slow)
libufdt	~10 ms (fast 🚀)

So **use libufdt** instead of older tools like FreeBSD-style DTO.

🧠 3. API of libufdt

```c
struct fdt_header *ufdt_apply_overlay(
        struct fdt_header *main_fdt_header,
        size_t main_fdt_size,
        void *overlay_fdt,
        size_t overlay_size);
```

🔧 What this does:

- Takes main DT and overlay DT
- Applies overlay
- Returns merged DT (`fdt_header*`)
- Use `dto_malloc()` internally to allocate memory

🚫 4. Can't Overlay Root Node

You **cannot add or change nodes at the root level (`/`)**.

❓ Why?

Because overlaying works **only if labels exist**, and the root node **can't have a label**.

✅ Workarounds:

1. **Use `odm` node**:

```dts
/ {
  odm: odm { }; // Base DT

  &odm {
    my_overlay_node { ... }; // Overlay DT
  }
};
```

2. **Group SoC things under a `soc` node**, so root remains untouched.

📦 5. Compressed DTBO Support (Android 9+)

✨ Why compress?

- To save space in flash
- Faster boot time

🔧 Where is it used?

In **DTBO partition** when using **dt_table_header version 1**

Format:

```c
struct dt_table_entry_v1 {
  ...
  uint32_t flags; // Lowest 4 bits tell compression
};
```

Supported compressions:

```c
enum dt_compression_info {
  NO_COMPRESSION,
  ZLIB_COMPRESSION,
  GZIP_COMPRESSION
};
```

✅ Use `mkdtimg` to generate compressed DTBOs.

Example:

```bash
mkdtimg create dtbo.img board1.dtbo board2.dtbo
```

🧪 6. Sample DTO Implementation (with libufdt)

```c
#include <libfdt.h>
#include <ufdt_overlay.h>

{
  struct fdt_header *main_fdt_header;
  struct fdt_header *merged_fdt;

  // 1. Load .dtb and .dtbo into memory
  main_size = my_load_main_dtb(main_buf, main_buf_size);
  overlay_size = my_load_overlay_dtb(overlay_buf, overlay_buf_size);

  // 2. Convert main DTB into UFDT (un-flattened tree)
  main_fdt_header = ufdt_install_blob(main_buf, main_size);

  // 3. Apply overlay
  merged_fdt = ufdt_apply_overlay(main_fdt_header, main_size,
                                  overlay_buf, overlay_size);

  // 4. Get final merged size
  merged_fdt_size = dtc_totalsize(merged_fdt);

  // 5. Pass merged DT to kernel
  my_kernel_entry(0, machine_type, merged_fdt);
}
```

✅ Summary Table

Topic	Summary
`bootargs` vs **`bootargs_ext`**	Bootloader must concatenate both
`libufdt`	Fast overlay engine for .dtb/.dtbo
Root overlay restriction	Root node can't be modified; use **`odm`** or **`soc`**
Compression	ZLIB/GZIP supported from Android 9
DTO code sample	Shows how to load, overlay, and boot

Would you like me to make a **flowchart diagram** of the DTO process using libufdt or compressed DTBOs?

25. ↪
libufdt
explain in simPLE WORDS

Sure! Let's explain **libufdt** in **very simple words**:

🧠 What is `libufdt`?

`libufdt` is a **software tool** that helps your bootloader **combine** a base device tree (`.dtb`) and an overlay device tree (`.dtbo`) **very fast**.

🤔 Why do we need to "combine" .dtb and .dtbo?

- The **.dtb** file describes your base hardware (like CPU, RAM, etc.)
- The **.dtbo** file adds or changes some hardware settings (like touchscreen, extra sensors, etc.)
- We need to **merge them** into one final file before booting Android

---

🐢 Problem with old method (libfdt):

- The old tool called `libfdt` is **slow**
- On big devices like Pixel, it could take **100 ms or more**

🚀 Solution: Use `libufdt`

- `libufdt` is a newer, smarter library from Google
- It's **10x faster** than `libfdt`
- Used in Pixel and modern Android devices

🔧 How it works

Here's how your bootloader should use `libufdt`:

1. **Load the base DTB** (`main.dtb`) from storage
2. **Load the overlay DTBO** (`board-specific.dtbo`) from storage
3. **Call `libufdt` to merge them**
4. **Give the merged result to the Linux kernel to boot**

🧪 Sample simple code (in C)

```c
#include <libfdt.h>
#include <ufdt_overlay.h>

// Load base and overlay
main_size = load("main.dtb", main_buf);
overlay_size = load("overlay.dtbo", overlay_buf);

// Prepare for overlay
main_tree = ufdt_install_blob(main_buf, main_size);

// Merge them
merged_tree = ufdt_apply_overlay(main_tree, main_size, overlay_buf, overlay_size);

// Boot with merged DT
boot_kernel(merged_tree);
```

📦 Summary (one-liner)

`libufdt` is a fast tool that merges your base device tree and overlay, so Android boots with the right hardware settings — quickly and reliably.

Would you like me to draw a **small diagram** for this?

26. What next should I read now

Great! Since you've now understood:

- ✅ What Device Tree Overlays (DTO) are
- ✅ How to compile and verify `.dts`, `.dtbo`
- ✅ How to use `libufdt` and optimize DTO in Android bootloader

🔁 Now, here's what you should read next, based on your current progress:

📘 1. Android Boot Flow and DTO Integration

Learn **where** and **how** DTBOs are used during Android boot.

- 📚 Topics:
  - Android boot stages: **Bootloader** → **DTBO load** → **Kernel**
  - `boot.img`, `dtbo.img`, `vendor_boot.img` structure
  - How `androidboot.dtbo_idx` works
  - When `mkbootimg` and `mkdtimg` are used

🧠 2. `mkbootimg` and `mkdtimg` Tools Deep Dive

You already touched `mkdtimg`. Now understand:

- 🔨 How to **create `boot.img` with `dtbo.img`** included
- How `mkbootimg` packs kernel + ramdisk + dtb + dtbo (Android 11+ uses vendor_boot.img)
- How AOSP build system generates these images
- Practice: generate your own DTBO image and flash it

⚙️ 3. Vendor Boot Partition (Android 10+)

Understand the `vendor_boot.img` (replaced old `boot.img` roles in modern AOSP):

- Fields inside: `ramdisk`, `dtb`, `dtbo` pointers
- Format versioning (1, 2, 3...)
- How vendor boot handles **modular kernel components**

🔩 4. How DTs relate to Android HALs and Drivers

- Once device trees describe the hardware...
- HALs and kernel drivers use that info!
- Learn:
  - How DT maps to Linux kernel driver
  - How HALs detect and talk to the hardware
  - Real example: touchscreen, LEDs, camera DT entries → HAL

🛠️ 5. Hands-on: Build AOSP with Custom DTBO

Try building AOSP with:

- Custom `main.dts`
- A separate `overlay.dts` file
- Generate `dtbo.img`
- Flash and boot in **Cuttlefish/QEMU** or real board (e.g., Raspberry Pi, Pixel dev board)

Would you like a **step-by-step learning roadmap** for this? I can organize it like a syllabus and suggest which files to explore in AOSP too.