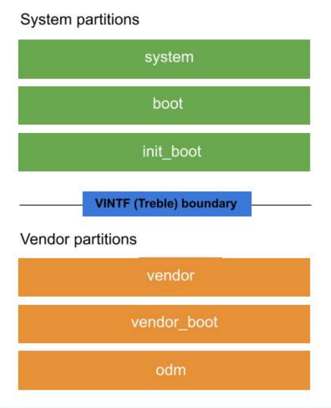
Android Partitions

1. Partitions overview bookmark_border Android devices contain several partitions or specific sections of storage space used to contain specific parts of the device's software. Each partition contains a partition image (an IMG file) or snapshot of all the software for the partition. Figure 1 shows the layout of core partitions on a device: Partitions are classified in three categories: System partitions are partitions that are updated when updating the OS and other features. The system, boot, and init_boot are core system partitions. Vendor partitions contain device and hardware-specific code that might never be updated after initial release. The vendor, vendor_boot, and odm partitions are core vendor partitions. Nonupdatable partitions. 3
2. Nonupdatable partitions Following is a list of all nonupdatable partitions and their use: cache partition. This partition contains temporary data and is optional if your device uses seamless updates. This partition doesn't need to be writable from the bootloader, but needs to be erasable. The partition size depends on the device type and the availability of space on userdata; typically, 50 to 100 MB is sufficient. userdata partition. This partition contains user-installed apps and data, including customization data. metadata partition. If your device uses metadata encryption, this partition contains the metadata encryption key. The size of this partition is 16 MB or larger, it isn't encrypted, and its data isn't snapshotted. This partiti. 8
3. Would you like a visual diagram for this section too? Or should we explore: * How to **check these partitions** on a real device? * How **dynamic partitioning works with `super.img`**? * Or how **Soong variants affect custom HAL or system modules**?explain these. 11
4. How VINTF works. 14
5. Partition layout bookmark_border In Android 10, the root file system is no longer included in ramdisk.img and is instead merged into system.img (that is, system.img is always created as if BOARD_BUILD_SYSTEM_ROOT_IMAGE was set). Devices launching with Android 10: Use a system-as-root partition layout (automatically enforced by the build with no options to change the behavior). Must use a ramdisk, which is required for dm-linear. Must set BOARD_BUILD_SYSTEM_ROOT_IMAGE to false. This setting is used only to differentiate between devices that use a ramdisk and devices that don't use a ramdisk (and instead mount system.img directly). The meaning of a system-as-root configuration differs between Android 9 and Android 10. In an Android 9 system. 17
6. what is rootfs explain in simple words with example. 23
7. why move from boot.img to system.img. 25
8. dynamic partition explain in simple words with example. 27
9. which block device super.img uses. 30
10. dm-verity explain in simple words with example. 32
11. dm-verity vs avb. 34
12. dm-linear. 36
13. lvm, fs_mgr, and update engine. 38
14. merkle tree. 40
15. Vendor boot partitions bookmark_border Android 11 introduced the concept of the Generic Kernel Image (GKI). To enable booting an arbitrary device with the GKI, Android 11 devices can use boot image header version 3. In version 3, all vendor-specific information is factored out of the boot partition and relocated into a new vendor_boot partition. An ARM64 device launching with Android 11 on the 5.4 Linux kernel must support the vendor_boot partition and the updated boot partition format to pass testing with the GKI. Android 12 devices can use boot image header version 4, which supports including multiple vendor ramdisks in the vendor_boot partition. Multiple vendor ramdisk fragments are concatenated one after another in the vendor ramdisk. 42
16. \[Bootloader\] ↓ Load vendor_boot.img: - All vendor ramdisks (based on mode: normal or recovery) - DTBs - Kernel modules ↓ Load boot.img: - Kernel - Generic ramdisk ↓ \[Kernel decompresses both ramdisks → creates one merged initramfs\] ↓ /init launches Androidwhy vendor_boot.img first then boot.img here. 48
17. Vendor and ODM DLKM partitions bookmark_border Android 11 introduces the concept of the Generic Kernel Image and the vendor boot partition. The vendor boot partition stores kernel modules compatible with GKI, and is loaded by first stage init. Kernel Modules prior to Android 11's release are also stored in vendor and ODM partitions, and are loaded by vendor processes. For Android 11 or higher, the kernel and all kernel modules can be updated independently from the rest of the partitions. To enable updates for kernel modules stored in the vendor partition (without a vendor partition update), move all vendor partition modules to a new partition called Vendor DLKM (dynamically loadable kernel module). You can then update this partition indep. 50
18. `fstab`explain simple words. 54
19. yes. 56
20. Android shared system image bookmark_border This page presents several mechanisms that Android device OEMs can use to have their own shared system image (SSI) across product lines. It also proposes a procedure for basing an OEM-owned SSI on an AOSP-built generic system image (GSI). Background With Project Treble, monolithic Android was split into two parts: the hardware-specific part (the vendor implementation) and the generic OS part (the Android OS framework). The software for each is installed in a separate partition: the vendor partition for the hardware-specific software, and the system partition for the generic OS software. A versioned interface, called the vendor interface (VINTF), is defined and enforced across the two partitions.. 58
21. it came after treble?. 61
22. Partitions around SSI Figure 1 shows partitions around SSI, and the versioned interfaces across the partitions and policies on the interfaces. This section explains each of the partitions and interfaces in detail. Images and partitions The information in this section distinguishes between the terms image and partition. An image is a conceptual piece of software that can be updated independently. A partition is a physical storage location that can be updated independently. The sections in Figure 1 are defined as follows: SSI: The SSI is the image that’s common to an OEM, and can exist across multiple devices. It doesn't have any hardware-specific or product-specific components. Everything in a given SSI is, by definition, shared among al. 62
23. A generic system image (GSI) is the system image that’s built directly from AOSP. It’s used for the Treble compliance tests (for example, CTS-on-GSI) and as a reference platform that app developers can use to test the compatibility of their apps when they don’t have a real device running the required version of Android. OEMs can also use GSI to make their SSI. As explained in Images and partitions, SSI consists of the system image for the AOSP-defined components and the system_ext image for the OEM-defined components. When GSI is used as the system image, the OEM can focus on the system_ext image for the upgrade. This section provides a guide to OEMs who want to modularize their customizations into the /system_ext and /product partitions. 66
24. GSI vs SSI. 70
25. Over time, SSI can be made to **match GSI** completely.Explain this statement. 72
26. In what sense SSI helping OEM?. 73
27. Ramdisk partitions bookmark_border Note: This page applies to Android 11 and lower. For Android 12 and greater, see Generic boot partition. In Android 10 and 11, the first stage ramdisk contains the first stage init binary (which performs early mounting as specified by fstab entries) and vendor fstab files. (As in Android 9 and lower, system.img contains the contents of $TARGET_ROOT_OUT.) For devices with a boot-ramdisk (non-A/B), first stage init is a static executable located at /init. These devices mount system.img as /system, then perform a switch root operation to move the mount at /system to /. The contents of the ramdisk are freed after mounting has completed. For devices that use recovery as a ramdisk, first stage init is located. 75
28. Confusing. 79
29. Generic boot partition bookmark_border In Android 12, the generic boot image, referred to as Generic Kernel Image (GKI), contains the generic ramdisk and the GKI kernel. For devices launching with Android 13, the generic ramdisk is removed from the boot image and placed in a separate init_boot image. This change leaves the boot image with only the GKI kernel. For upgrading devices that continue to use Android 12 or older kernel versions, the generic ramdisk remains where it was with no requirement for a new init_boot image. To build a generic ramdisk, move vendor-specific resources out of the ramdisk such that the generic ramdisk contains only first stage init and a property file that contains timestamp information. On devices that: D. 81

30. so there is different boot flow in recovery mode? 94
31. can a device can be A/B and non A/B? 96
32. means A/B is feature? 98
33. how to check a device support A/B. 99
34. ODM partitions bookmark_border Android 10 includes support for building odm partitions using the Android build system. About ODM partitions Original design manufacturers (ODMs) customize system-on-chip (SoC) vendor board-support packages (BSPs) to their specific devices (their boards). This enables them to implement kernel modules for board-specific components, board-specific daemons, or their own features on hardware abstraction layers (HALs). They might also want to replace or customize SoC components. In lower Android releases, such customizations prevented the use of a single vendor image for devices with the same SoC (or with different SoCs, but in the same family). In Android 10 and higher, you can use a separate odm partition for. 101
35. Product partitions bookmark_border Android 9 and higher includes support for building product partitions using the Android build system. Previously, Android 8.x enforced the separation of SoC-specific components from the system partition to the vendor partition without dedicating space for OEM-specific components built from the Android build system. Android 9 and higher provides additional permissions and whitelisting features that apply to priv-apps on different partitions. About product partitions Many OEMs customize the AOSP system image to implement their own features, as well as carrier requirements. However, such customizations make it impossible to use a single system image for multiple software SKUs. Each image must be different t. 104
36. YES. 108
37. what is the use of interfaces between these partitions and why it exist. 109
38. Implement a GKI module partition bookmark_border GKI and GKI modules can be updated independently from the rest of the partition because GKI modules reside on a separate dynamic partition in the super image called system_dlkm. GKI modules are signed by Google using the kernel build-time key pair and are compatible only with the GKI they're built with. There's no ABI stability between GKI and GKI modules; for modules to load correctly during runtime, GKI and GKI modules must be built and updated together. Note: Any device launching with Android 13 and higher must include a system_dlkm partition. Implement system_dklm partition support The system_dlkm partition is located in the super partition as another dynamic partition. This partition c. 111
39. `system_dlkm`why it is required and when introduced. 114
40. Enforce product partition interfaces bookmark_border Android 11 unbundles the product partition, making it independent of the system and vendor partitions. As part of these changes, you can now control the product partition's access to native and Java interfaces (which is similar to how interface enforcement works for vendor partitions). Enforce native interfaces To enable the native interface enforcement, set PRODUCT_PRODUCT_VNDK_VERSION to current. (The version is automatically set to current when the shipping API level for the target is greater than 29.) Enforcement allows: Native modules in the product partition to link: Statically or dynamically to other modules in the product partition that include static, shared, or header librari. 116
41. is this interface enforcement exist both way ?. 122
42. can you explain this enforcement in simple words with examples. 123
43. **step-by-step checklist** to apply enforcement on your own custom device.yes please. 125
44. ### 🔹 **\[7\] Enforce Artifact Path Separation (Optional)** **Goal**: Ensure only system modules go into `/system`, others to `/product`, `/vendor`, etc. #### ✅ In `device.mk`: makefile CopyEdit `PRODUCT_ENFORCE_ARTIFACT_PATH_REQUIREMENTS := strict` #### ✅ Define allowed install paths: makefile CopyEdit `_my_paths := $(TARGET_COPY_OUT_ROOT)/ $(TARGET_COPY_OUT_SYSTEM)/ $(call require-artifacts-in-path, $(_my_paths),)`why this. 129
45. Trusty OS (TOS) partitions bookmark_border Trusty is Google's implementation of a Trusted Execution Environment (TEE) OS that runs alongside Android. This is the specification for devices using Arm Trustzone technology to provide a TEE. If your ARM devices use Trusty as the secure OS solution, implement the bootloader as described in the following sections. Initialize TOS To load and initialize the Trusty OS (TOS), a bootloader should: Set up and configure all available RAM. Initialize at least one serial port. Verify signature of TOS image. Load TOS into RAM (execution from flash or TCM isn't supported). Jump to the first instruction in the TOS image after setting up the state and registers as described below. Call in to the TOS image C. 131
46. what's the job of Trusty OS explain in simple words. 134
47. which will be loaded first Trusty OS or Android. 136
48. map the partitions with aosp code directories. 137
49. why in some cases one partition has mapping to multiple directories?. 138
50. a same directory or any can belong to many partitions?. 140
51. how to map these directories to these partitions tell for all partitions please create table for this. 142

1.



Partitions overview

bookmark_border
Android devices contain several partitions or specific sections of storage space used to contain specific parts of the device's software. Each partition contains a partition image (an IMG file) or snapshot of all the software for the partition. Figure 1 shows the layout of core partitions on a device:

Partitions are classified in three categories:

System partitions are partitions that are updated when updating the OS and other features. The system, boot, and init_boot are core system partitions.

Vendor partitions contain device and hardware-specific code that might never be updated after initial release. The vendor, vendor_boot, and odm partitions are core vendor partitions.

Nonupdatable partitions are partitions whose contents are either not updated or updated with user data.

Code in system and vendor partitions can interact using a stable interface called the vendor interface (VINTF).

Note: The separation of system partitions from vendor partitions was part of an Android 11 effort called Project Treble. With this architecture, you can update a device's operating system and apps without updating any of hardware-specific code.
System partitions
Following is a list of all system partitions and their use:

boot partition. This partition contains a Generic Kernel Image (GKI). This partition also contains the generic ramdisk in devices launched in Android 12 and lower. For further information on generic ramdisk, see Generic ramdisk image contents.

init_boot partition (Android 13 and higher). This partition contains a generic ramdisk. In Android 11 and 12, the generic ramdisk is in the boot partition.

system partition. This partition contains the system image used for OEM products.

system_ext partition. This partition contains system resources and proprietary system modules that extend the common system image in the system partition.

Note: Single system image (SSI) refers to a file, such as a zip file that contains the images of the system and system_ext partitions and reuses those images across a set of target devices. For further information on SSI, see Android shared system image.
system_dlkm partition. This partition contains GKI modules. For further information on this partition, see Implement a GKI module partition.

product partition. This partition can contain product-specific modules that aren't bundled with any other partitions.

Note: The Vendor Native Development Kit (VNDK) is a set of libraries installed in the system partition and designed exclusively for vendors to implement their HALs. The product and vendor partitions can link to VNDK libraries in the system partition, but can't link to other libraries in the system partition.
pvmfw partition. This partition stores the Protected Virtual Machine Firmware (pvmfw) which is the first code that runs in protected VMs. For more information, see Protected Virtual Machine Firmware.

generic_bootloader partition. This partition contains the generic bootloader.

Vendor partitions
Following is a list of all vendor partitions and their use:

vendor_boot partition. This partition contains vendor-specific boot code. For more information, see Vendor boot partitions.

recovery partition. This partition stores the recovery image, which is booted during the over-the-air (OTA) update process. Devices that support seamless updates can store the recovery images as a ramdisk contained in the boot or init_boot image. For more information on seamless updates, see A/B (seamless) updates.

vbmeta partition. This partition contains the Verified Boot information for all of the partitions. This information verifies that the images installed in each partition is trusted. For further information on Verified Boot, see Verified Boot.

vendor partition. This partition contains any binary that is vendor specific and not generic enough to distribute to AOSP.

Note: The Vendor Native Development Kit (VNDK) is a set of libraries installed in the system partition and designed exclusively for vendors to implement their HALs. The product and vendor partitions can link to VNDK libraries in the system partition, but can't link to other libraries in the system partition.
vendor_dlkm partition. This partition contains vendor kernel modules. By storing vendor kernel modules in this partition instead of the vendor partition, you can update kernel modules without updating the vendor partition. For more information, see Vendor and ODM DKLM partitions.

odm partition. This partition contains original design manufacturer (ODM) customizations to system-on-chip (SoC) vendor board-support packages (BSPs). Such customizations enable ODMs to replace or customize SoC components, and implement kernel modules for board-specific components, daemons, and ODM-specific features on hardware abstraction layers (HALs). This partition is optional. Typically this partition is used to contain customizations so that devices can use a single vendor image for multiple hardware SKUs. For more information, see ODM partitions.

odm_dlkm partition. This partition is dedicated to storing ODM kernel modules. By storing ODM kernel modules in the this partition, instead of the odm partition, you can update ODM kernel modules without updating the odm partition. For more information, see Vendor and ODM DKLM partitions.

radio partition. This partition contains the radio image and is needed only for devices that include a radio with radio-specific software in a dedicated partition.

Note: Devices that support seamless updates need two partitions, referred to as slots (slot A and slot B) for the boot, system, vendor, and radio partitions. For further information, see Partition selection (slots).
Nonupdatable partitions
Following is a list of all nonupdatable partitions and their use:

cache partition. This partition contains temporary data and is optional if your device uses seamless updates. This partition doesn't need to be writable from the bootloader, but needs to be erasable. The partition size depends on the device type and the availability of space on userdata; typically, 50 to 100 MB is sufficient.

userdata partition. This partition contains user-installed apps and data, including customization data.

metadata partition. If your device uses metadata encryption, this partition contains the metadata encryption key. The size of this partition is 16 MB or larger, it isn't encrypted, and its data isn't snapshotted. This partition is erased when the device is factory reset.

Partition update rules and recommendations
We recommend updating all system partitions as a whole and all vendor partitions as another whole. By updating the set of partitions as a whole, you can test to verify the interfaces between images in each partition remain stable.

Regardless of how you update your partitions, the following partitions must be updated due to tightly coupled dependencies and lack of stable APIs:

The boot and system_dlkm partitions
the init_boot, system, system_ext, and product partitions
Note: If all interfaces between the product partition and other system partitions have stable ABIs, you can update the product partition independently. For furthe information, see Maintain ABIs between partitions.
Dynamic partitions
Devices running Android 11 and higher can support dynamic partitions, which are a userspace partitioning system for Android that lets you create, resize, or destroy partitions during over-the-air (OTA) updates. For more information, see Dynamic partitions.

misc partition. This partition is used by the recovery partition and is 4 KB or larger.

tos partition. This partition contains the binary image of the Trusty OS and is used only if the device includes Trusty. For more information, see Trusty OS (TOS) partitions.

Soong product variants
The Soong build system uses image variants to split build dependencies. Native modules (/build/soong/cc) can mutate system process modules to the core variant and vendor process modules to the vendor variant; a module in one image variant can't link to other modules in a different image variant.

In Android 12 or higher, a system module with vendor_available: true creates a vendor variant in addition to the core variant. To create a product variant, product_available: true must be defined. Some VNDK libraries without product_available: true aren't available to product modules.

Was this helpful?

Thanks! Based on the **diagram** (`core-partitions.png`) and your **provided documentation**, here's a **simple breakdown** of Android's core partitions:

🟩 System Partitions (OS + Features - Updatable)

These contain the **generic OS code** and are updated with Android updates.

- `system`: Main Android OS code and apps.
- `boot`: Contains the **Generic Kernel Image (GKI)** and (on Android 12 or lower) the **generic ramdisk**.
- `init_boot`: (Android 13+) Contains the **generic ramdisk** (moved here from `boot`).
- ➕ (not in image but part of system):
  - `system_ext`: OEM/system vendor extensions to `system`.
  - `product`: Device-specific system features.
  - `system_dlkm`: GKI kernel modules.

🔵 VINTF (Treble) Boundary

This is a **strict separation line** between system and vendor partitions:

- Ensures system updates can happen **independently** of vendor code.
- Interfaces defined in **VINTF manifest** must stay **stable**.

🟧 Vendor Partitions (Hardware-specific - Less frequently updated)

These are tailored for the **specific device hardware**.

- `vendor`: Contains hardware drivers, HALs, binaries.
- `vendor_boot`: Boot components needed by vendor-specific hardware.
- `odm`: OEM/ODM customizations (optional).
- ➕ (not shown in the image but related):
  - `vendor_dlkm`: Vendor kernel modules (separate from `vendor` for modular updates).
  - `odm_dlkm`: Kernel modules specifically for ODM.
  - `radio`: Cellular modem firmware (if present).
  - `vbmeta`: Verified Boot metadata (verifies partition integrity).

⚫ Non-Updatable Partitions

- `userdata`: Stores user apps and data.
- `cache`: Temporary storage.
- `metadata`: Encryption metadata.
- `misc`, `tos`, etc.: Miscellaneous special-purpose partitions.

💡 Summary of How It All Works:

- **System** = Google's Android (updated with Android updates)
- **Vendor** = Manufacturer's hardware support (rarely updated)
- **Treble/VINTF** = Stable interface that separates them
- Ensures faster Android updates without needing to update vendor HALs or drivers.

If you'd like, I can also:

- Compare this layout to **A/B partitions**
- Explain **dynamic partitions**
- Show **real partition layout** from a device (`ls -l /dev/block/by-name/`)
- Or help you write a script to inspect them on Android

Let me know what you'd like to dive into next.
Sure! Let's break it down **in simple language** so you can understand what each Android partition does and why it's there.

✅ What Are Partitions?

Think of **partitions** like different sections of a hard disk.
Each one stores a specific part of Android—like the system, hardware drivers, or your personal data.

🟩 System Partitions (Green ones in the image)

These are parts of Android that **Google or the phone company updates**.
They make your Android OS run.

1. `system`
    - Main Android operating system (apps, UI, settings).
2. `boot`
    - Contains the **kernel** (like the engine of Android).
    - Also had ramdisk (tiny OS to start Android) in older Android (up to version 12).
3. `init_boot`
    - New in Android 13+
    - Stores the **ramdisk** (now moved out of `boot`).

👉 These parts are updated during a software update.

🔵 VINTF (Treble) Boundary

Think of this as a **fence** between:

- Android OS (`system`, `boot`)
- Hardware stuff (`vendor`, `odm`, etc.)

Why?
So Google can update Android **without needing to touch hardware code.**
This is called **Project Treble** (introduced in Android 8, improved in Android 11+).

🟧 Vendor Partitions (Orange ones)

These belong to **your device maker (e.g., Samsung, Xiaomi)**.
They contain drivers and software to control your hardware (camera, screen, sensors, etc.).

1. `vendor`

   - Drivers and HALs (Hardware Abstraction Layers) for your phone.
2. `vendor_boot`
   - Special files needed to boot your hardware.
3. `odm`
   - Custom hardware code by the OEM/ODM (Original Device Manufacturer).
     Example: Mic in India might need changes compared to a global version.

👉 These are **rarely updated**.

⚫ Other Partitions (Not in the diagram but important)

1. `userdata`
   - Your apps, files, photos, etc.
2. `cache`
   - Temporary data like download files or app updates.
3. `vbmeta`
   - Verifies if your system hasn't been hacked (Verified Boot).
4. `metadata`
   - Holds keys if your data is encrypted.
5. `recovery`
   - Used for factory reset or system update mode.

🧠 Why So Many Partitions?

To keep things **modular and safe**:

- System can update without touching vendor drivers.
- You can **wipe userdata** without touching system files.
- Hardware drivers stay **stable and separate**.

📱 Real-life Example

When your phone gets an Android update:

- `system`, `boot`, `init_boot` partitions are updated.
- `vendor` stays the same unless a major update is needed.
- Your `userdata` (apps/photos) remains safe.

Would you like:

- A **real command** to see partitions on your phone?
- Or more explanation on **dynamic partitions**, A/B system, or boot process?

Let me know!

2. Nonupdatable partitions
Following is a list of all nonupdatable partitions and their use:

cache partition. This partition contains temporary data and is optional if your device uses seamless updates. This partition doesn't need to be writable from the bootloader, but needs to be erasable. The partition size depends on the device type and the availability of space on userdata; typically, 50 to 100 MB is sufficient.

userdata partition. This partition contains user-installed apps and data, including customization data.

metadata partition. If your device uses metadata encryption, this partition contains the metadata encryption key. The size of this partition is 16 MB or larger, it isn't encrypted, and its data isn't snapshotted. This partition is erased when the device is factory reset.

Partition update rules and recommendations
We recommend updating all system partitions as a whole and all vendor partitions as another whole. By updating the set of partitions as a whole, you can test to verify the interfaces between images in each partition remain stable.

Regardless of how you update your partitions, the following partitions must be updated due to tightly coupled dependencies and lack of stable APIs:

The boot and system_dlkm partitions
the init_boot, system, system_ext, and product partitions
Note: If all interfaces between the product partition and other system partitions have stable ABIs, you can update the product partition independently. For furthe information, see Maintain ABIs between partitions.
Dynamic partitions
Devices running Android 11 and higher can support dynamic partitions, which are a userspace partitioning system for Android that lets you create, resize, or destroy partitions during over-the-air (OTA) updates. For more information, see Dynamic partitions.

misc partition. This partition is used by the recovery partition and is 4 KB or larger.

tos partition. This partition contains the binary image of the Trusty OS and is used only if the device includes Trusty. For more information, see Trusty OS (TOS) partitions.

Soong product variants
The Soong build system uses image variants to split build dependencies. Native modules (/build/soong/cc) can mutate system process modules to the core variant and vendor process modules to the vendor variant; a module in one image variant can't link to other modules in a different image variant.

In Android 12 or higher, a system module with vendor_available: true creates a vendor variant in addition to the core variant. To create a product variant, product_available: true must be defined. Some VNDK libraries without product_available: true aren't available to product modules.

Let’s now simplify this section in **plain words** so it’s easy to understand, especially for Android development or kernel work:

⚫ Nonupdatable Partitions (Can't Be Updated by OTA System Images)

These partitions store **temporary or user-specific data**, not part of the OS or hardware drivers.

1. `cache`

- Stores **temporary system data**.
- Example: App updates, logs, or files downloaded during OTA.
- Not needed if the device supports **A/B seamless updates**.
- Typically **50–100 MB**.
- Can be erased and recreated anytime.

---

2. `userdata`

- This is **your personal data partition**:
  - Installed apps

  - App data
  - Downloads, images, etc.
- Gets wiped when doing a **factory reset**.
- It's the **largest partition** on most phones.

---

3. `metadata`

- Used for **encryption support** (like file-based encryption).
- Holds keys and metadata for secure storage.
- Around **16MB or more**.
- Itself is **not encrypted**, but stores data needed for encryption.
- Also wiped during a factory reset.

🔁 Partition Update Rules (Who Should Be Updated Together?)

To avoid crashing or boot failures:

- **System partitions** (`system`, `boot`, `init_boot`, etc.) should be updated **together**.
- **Vendor partitions** (`vendor`, `vendor_boot`, `odm`) should be updated **together**.
- This keeps the **interface between them stable** (called VINTF compatibility).

Must-be-updated-together examples:

- `boot` + `system_dlkm`
- `init_boot`, `system`, `system_ext`, and `product`

🧱 Dynamic Partitions

Android 11+ supports **dynamic partitions**, which are:

- **Flexible partitions** created in **userspace**.
- Can be **resized, added, or removed** during an OTA update.
- Help in customizing different devices **without flashing a new full image**.

No need to fix partition sizes in advance.

Example:

```bash
super.img → contains system, vendor, product as logical partitions
```

🧩 Other Special Partitions

1. `misc`

- Small partition (≥4 KB).
- Used by the **recovery system** to store update status, boot modes, etc.

2. `tos`

- Stores **Trusty OS**, which is a secure, isolated OS inside Android.
- Used for **security features** like keystore, biometrics, etc.
- Only needed on devices using Trusty.

🏗️ Soong Product Variants (Build System Details)

Android's build system (`Soong`) builds modules in **different image variants**:

Variant	Purpose
`core`	For system image (generic Android)
`vendor`	For vendor-specific code
`product`	For product-specific modules

Key Flags:

- `vendor_available: true`: Makes module available to vendor variant.
- `product_available: true`: Makes module available to product variant.
- Some VNDK libs might not be available unless explicitly marked.

🔒 **Rule**: Modules in one variant **cannot link** to modules in a different variant.

✅ Summary Table

Partition	Type	Description
`cache`	Non-updatable	Temporary files
`userdata`	Non-updatable	Apps, photos, settings
`metadata`	Non-updatable	Encryption-related info
`misc`	Special	Recovery control
`tos`	Special	Trusty secure OS
`system` etc.	Updatable	Android OS
`vendor` etc.	Updatable	Hardware-specific code

Would you like a visual diagram for this section too? Or should we explore:

- How to **check these partitions** on a real device?
- How **dynamic partitioning works with** `super.img`?
- Or how **Soong variants affect custom HAL or system modules**?

Let me know your next interest!

3. ⤷
Would you like a visual diagram for this section too? Or should we explore: * How to **check these partitions** on a real device? * How **dynamic partitioning works with `super.img`**? * Or how **Soong variants affect custom HAL or system modules**?
explain these

Perfect! Let's go through **all three** step by step in **simple language** — with examples and use-cases for each.

🔍 1. How to Check Partitions on a Real Android Device

✅ Method 1: Using ADB shell

```bash
adb shell
```

Then run:

```bash
ls -l /dev/block/by-name/
```

This will list all the **named partitions** like:

```perl
cache -> /dev/block/mmcblk0p25
system -> /dev/block/mmcblk0p28
userdata -> /dev/block/mmcblk0p32
vendor -> /dev/block/mmcblk0p29
```

Each symbolic link points to an actual partition block device.

✅ Method 2: Get partition size and mount info

```bash
cat /proc/partitions
cat /proc/mounts
```

✅ Example

```bash
# View system partition mount
mount | grep system

# Check userdata usage
df -h /data
```

You can also dump partition table using:

```bash
cat /proc/emmc
```

or

```bash
cat /proc/dumchar_info  # For MediaTek-based devices
```

💡 2. How Dynamic Partitioning Works (`super.img`)

🚀 What is `super.img`?

In Android 10+, dynamic partitioning was introduced. Instead of flashing fixed partitions, Android creates **logical partitions inside a shared "super" partition**.

```
bash

super.img = container for system, vendor, product, etc.
```

✅ Benefits:

- No need to predefine sizes for each partition.
- You can **resize, create, or delete** logical partitions during OTA updates.
- Helps OEMs ship fewer images for more devices.

✅ Check dynamic partitions:

```
bash

adb shell
cat /proc/mounts | grep /dev/block/mapper
```

You'll see:

```
bash

/dev/block/mapper/system_a  /system
/dev/block/mapper/vendor_a  /vendor
```

These are **logical partitions**, created dynamically from **`super.img`**.

🏗️ 3. Soong Image Variants & Build Rules

📦 What is Soong?

Soong is the Android build system (replacing Make). It builds modules like `.so` libraries, `.apk`s, etc.

🔄 Variants:

Every module can have multiple **variants** based on where it will go:

Variant	Use Case
core	Goes in **`system.img`** (Android generic code)
vendor	Goes in **`vendor.img`** (hardware-specific HALs, drivers)
product	Goes in **`product.img`** (OEM custom features)

🔧 Important flags in `Android.bp`

Flag	Meaning
`vendor_available: true`	Module is also built for vendor variant
`product_available: true`	Module is also built for product variant
`soc_specific: true`	Vendor-specific, not reused
`proprietary: true`	Closed-source

❌ Linking rules:

> Modules **cannot link** across variants.

For example:

- **`vendor`** module **can link** to **`VNDK`** (subset of system)
- But **`vendor`** module **cannot link** to full system libraries

This is enforced by the build system to maintain **Treble compatibility**.

✅ Example:

If you're writing a HAL:

```
c++

cc_library {
    name: "libcustomhal",
    srcs: ["custom_hal.cpp"],
    shared_libs: ["libhardware"],
    vendor: true,
}
```

This ensures:

- It gets installed in `vendor/`
- It only links with `vendor` or `VNDK` libraries

📌 Summary

Topic	Key Point
🔍 Check Partitions	Use `/dev/block/by-name/` or `/proc/mounts`
💡 Dynamic Partitions	Logical partitions inside `super.img`, flexible OTA
🏗 Soong Variants	Controls where module goes (`system`, `vendor`, `product`) and what it can link with

---

Would you like:

- A **diagram** showing how dynamic partitions work?
- A small **Soong `Android.bp` demo**?
- Or to learn how to **add a custom HAL to vendor**?

Let me know what you want to go deeper into!

4. How VINTF works

Great question! Let's **break down VINTF (Vendor Interface)** in simple terms — what it is, why it's needed, and how it works with diagrams and real-world examples.

🔷 What is VINTF?

VINTF = **Vendor Interface (Treble)**
It's a **contract** between:

- **Android OS** (system partition)
- **Vendor implementation** (vendor/odm partitions)

📦 Purpose:

VINTF **ensures compatibility** between the Android framework and device-specific HALs and drivers **even when only one side is updated**.

🔧 Why Do We Need It?

Before Android 8:

- HALs (hardware drivers) were tightly coupled to Android framework.
- You had to recompile HALs with every Android update.

This **slowed down updates**.

After Treble (Android 8+):

VINTF creates a **stable boundary** — a clear line — between system and vendor code.

Now:

- OEMs can **update Android OS** (system) without touching hardware code (vendor).
- HALs must use **stable interfaces**.

📘 Key VINTF Files

1. `compatibility_matrix.xml` (in system partition)

- Written by **Google** or Android build system
- Says: "I need these HALs, in these versions"

2. `manifest.xml` (in vendor partition)

- Written by the **device manufacturer**
- Says: "Here are the HALs I provide, and their versions"

🔄 How VINTF Works Step-by-Step

🔁 On Boot:

1. Android compares the **system's** `compatibility_matrix.xml`
2. With the **vendor's** `manifest.xml`
3. Uses `vintf_framework` service to **verify compatibility**

✅ If HAL versions match: boot continues
❌ If not: **boot fails** (with VINTF mismatch)

🔎 Simple Analogy

System (Google)	Vendor (OEM)	Compatible?
Wants `camera@3.5` HAL	Provides `camera@3.5`	✅ Yes
Wants `wifi@1.0` HAL	Provides `wifi@0.9`	❌ No (boot fails)

🧱 Real Files on Device

You can see them here:

```bash
adb shell
cat /vendor/manifest.xml
cat /system/compatibility_matrix.xml
```

Or from AOSP:

```bash
hardware/interfaces/compatibility_matrices/
```

📁 Example: `manifest.xml` (on device/vendor)

```xml
<manifest version="1.0" type="device">
  <hal>
    <name>android.hardware.camera</name>
    <version>3.5</version>
    <interface>
      <name>ICameraProvider</name>
      <instance>default</instance>
    </interface>
  </hal>
</manifest>
```

📁 Example: `compatibility_matrix.xml` (on system)

```xml
<compatibility-matrix version="1.0" type="framework">
  <hal>
    <name>android.hardware.camera</name>
    <version>3.5</version>
    <interface>
      <name>ICameraProvider</name>
      <instance>default</instance>
    </interface>
  </hal>
</compatibility-matrix>
```

✅ Benefits of VINTF

Feature	Benefit
Stable HAL versioning	No need to rebuild HALs with every update
Faster Android Updates	System can be updated without touching vendor
AOSP Compatibility	Easier to use a common GSI (Generic System Image)
Dynamic HAL loading	Only needed HALs are loaded

📊 Summary

Term	Location	Role
`manifest.xml`	`/vendor/`	Declares provided HALs
`compatibility_matrix.xml`	`/system/`	Declares required HALs
`VINTF`	Boundary	Validates match at boot

📌 Bonus: VINTF Tooling

You can run validation with:

```bash
lshal               # Lists available HALs
vintf_object --check-compat
```

Would you like:

- A **diagram** showing how the system and vendor interact via VINTF?
- A custom **HAL addition example with VINTF update**?
- Or how to create/update `manifest.xml` in AOSP?

Let me know!

5. Partition layout

bookmark_border
In Android 10, the root file system is no longer included in ramdisk.img and is instead merged into system.img (that is, system.img is always created as if BOARD_BUILD_SYSTEM_ROOT_IMAGE was set). Devices launching with Android 10:

Use a system-as-root partition layout (automatically enforced by the build with no options to change the behavior).
Must use a ramdisk, which is required for dm-linear.
Must set BOARD_BUILD_SYSTEM_ROOT_IMAGE to false. This setting is used only to differentiate between devices that use a ramdisk and devices that don't use a ramdisk (and instead mount system.img directly).
The meaning of a system-as-root configuration differs between Android 9 and Android 10. In an Android 9 system-as-root configuration, BOARD_BUILD_SYSTEM_ROOT_IMAGE is set to true, which forces the build to merge the root file system into system.img then mount system.img as the root file system (rootfs). This configuration is mandatory for devices launching with Android 9 but is optional for devices upgrading to Android 9 and for devices running lower versions of Android. In an Android 10 system-as-root configuration, the build always merges $TARGET_SYSTEM_OUT and $TARGET_ROOT_OUT into system.img; this config is the default behavior for all devices running Android 10.

Android 10 makes further changes to support dynamic partitions, a userspace partitioning system that enables over-the-air (OTA) updates to create, resize, or destroy partitions. As part of this change, the Linux kernel can no longer mount the logical system partition on devices running Android 10, so this operation is handled by the first stage init.

The following sections describe the system-as-root requirements for system-only OTAs, provide guidance on updating devices to use system-as-root (including partition layout changes and dm-verity kernel requirements). For details on changes to ramdisk, see Ramdisk Partitions.

About system-only OTAs
System-only OTAs, which enable Android releases to update system.img and product.img without changing other partitions, require a system-as-root partition layout. All devices running Android 10 must use a system-as-root partition layout to enable system-only OTAs.

A/B devices, which mount the system partition as rootfs, already use system-as-root and don't require changes to support system OTAs.
Non-A/B devices, which mount the system partition at /system, must be updated to use a system-as-root partition layout to support system OTAs.
For details on A/B and non-A/B devices, refer to A/B (Seamless) System Updates.

Use vendor overlay (<=AOSP 14)
Vendor overlay allows you to overlay changes to the vendor partition at device boot time. A vendor overlay is a set of vendor modules in the product partition that get overlaid on the vendor partition when the device boots, replacing and adding to the existing modules.

When the device boots, the init process completes the first stage mount and reads the default properties. Then it searches /product/vendor_overlay/<target_vendor_version> and mounts each subdirectory on its corresponding vendor partition directory, if the following conditions are met:

/vendor/<overlay_dir> exists.
/product/vendor_overlay/<target_vendor_version>/<overlay_dir> has the same file context as /vendor/<overlay_dir>.
init is allowed to mount on the file context of /vendor/<overlay_dir>.
Implement vendor overlay
Install vendor overlay files in /product/vendor_overlay/<target_vendor_version>. Those files overlay the vendor partition when the device boots, replacing files of the same name and adding any new files. Vendor overlay can't remove files from the vendor partition.

Vendor overlay files must have the same file context as the target files they replace in the vendor partition. By default, the files in the /product/vendor_overlay/<target_vendor_version> directory have the vendor_file context. If there are file context mismatches between vendor overlay files and the files they replace, specify that in the device-specific sepolicy. File context is set at the directory level. If the file context of a vendor overlay directory doesn't match the target directory, and the correct file context isn't specified in the device-specific sepolicy, that vendor overlay directory isn't overlaid onto the target directory.

To use vendor overlay, the kernel must enable OverlayFS by setting CONFIG_OVERLAY_FS=y. Also, the kernel must be merged from the common kernel 4.4 or later, or patched with "overlayfs: override_creds=off option bypass creator_cred".

Vendor overlay implementation example
This procedure demonstrates implementing a vendor overlay that overlays the directories /vendor/lib/*, /vendor/etc/*, and /vendor/app/*.

Add prebuilt vendor files in device/<vendor>/<target>/vendor_overlay/<target_vendor_version>/:

device/google/device/vendor_overlay/28/lib/libfoo.so
device/google/device/vendor_overlay/28/lib/libbar.so

```
device/google/device/vendor_overlay/28/etc/baz.xml
device/google/device/vendor_overlay/28/app/qux.apk
```

Install the prebuilt vendor files to product/vendor_overlay in device/google/device/device.mk:

```
PRODUCT_COPY_FILES += \
    $(call find-copy-subdir-files,*,device/google/device/vendor_overlay,$(TARGET_COPY_OUT_PRODUCT)/vendor_overlay)
```

Define file contexts if the target vendor partition files have contexts other than vendor_file. Because /vendor/lib/* uses the vendor_file context, this example doesn't include that directory.

Add the following to device/google/device-sepolicy/private/file_contexts:

```
/(product|system/product)/vendor_overlay/[0-9]+/etc(/.*)?   u:object_r:vendor_configs_file:s0
/(product|system/product)/vendor_overlay/[0-9]+/app(/.*)?   u:object_r:vendor_app_file:s0
```

Allow the init process to mount the vendor overlay on file contexts other than vendor_file. Because the init process already has permission to mount on the vendor_file context, this example doesn't define the policy for vendor_file.

Add the following to device/google/device-sepolicy/public/init.te:

```
allow init vendor_configs_file:dir mounton;
allow init vendor_app_file:dir mounton;
```

Validate vendor overlay

To validate the vendor overlay configuration, add files in /product/vendor_overlay/<target_vendor_version>/<overlay_dir> and check if the files are overlaid on the files in /vendor/<overlay_dir>.

For userdebug builds, there is a test module for Atest:

```
$ atest -v fs_mgr_vendor_overlay_test
```

Update to system-as-root

To update non-A/B devices to use system-as-root, you must update the partitioning scheme for boot.img and system.img, set up dm-verity, and remove any boot dependencies on the device-specific root folders.

Update partitions

Unlike A/B devices that repurpose /boot as the recovery partition, non-A/B devices must keep the /recovery partition separate as they don't have the fallback slot partition (for example, from boot_a to boot_b). If /recovery is removed on non-A/B device and made similar to the A/B scheme, recovery mode could break during a failed update to the /boot partition. For this reason, the /recovery partition must be a separate partition from /boot for non-A/B devices, which implies that the recovery image continues to be updated in a deferred manner (that is, the same as in devices running Android 8.1.0 or lower).

The following table lists image partition differences for non-A/B devices before and after Android 9.

Image	Ramdisk (before 9)	System-as-root (after 9)
boot.img	Contains a kernel and a ramdisk.img:

```
ramdisk.img
  -/
    - init.rc
    - init
    - etc -> /system/etc
    - system/ (mount point)
    - vendor/ (mount point)
    - odm/ (mount point)
    ...
```

Contains a normal boot kernel only.
recovery.img	Contains a recovery kernel and a recovery ramdisk.img.
system.img	Contains the following:

```
system.img
  -/
    - bin/
    - etc
    - vendor -> /vendor
    - ...
```

Contains the merged content of original system.img and ramdisk.img:

```
system.img
  -/
    - init.rc
    - init
    - etc -> /system/etc
    - system/
      - bin/
      - etc/
      - vendor -> /vendor
      - ...
    - vendor/ (mount point)
    - odm/ (mount point)
    ...
```

The partitions themselves don't change; both ramdisk and system-as-root use the following partition scheme:

/boot
/system
/system
/recovery
/vendor, etc.

Set up dm-verity
In system-as-root, the kernel must mount system.img under / (mount point) with dm-verity. AOSP supports the following dm-verity implementations for system.img.

vboot 1.0
For vboot 1.0, the kernel must parse Android-specific metadata on /system, then convert to dm-verity params to set up dm-verity (requires these kernel patches). The following example shows dm-verity related settings for system-as-root in kernel command line:

```
ro root=/dev/dm-0 rootwait skip_initramfs init=/init
dm="system none ro,0 1 android-verity /dev/sda34"
veritykeyid=id:7e4333f9bba00adfe0ede979e28ed1920492b40f
```

vboot 2.0
For vboot 2.0 (AVB), the bootloader must integrate external/avb/libavb, which then parses the hashtree descriptor for /system, converts it to dm-verity params, and finally passes the params to the kernel through the kernel command line. (Hashtree descriptors of /system might be on /vbmeta or on /system itself.)

vboot 2.0 requires the following kernel patches:

https://android-review.googlesource.com/#/c/kernel/common/+/158491/
kernel 4.4 patches, kernel 4.9 patches, etc.
Note: AVB-specific kernel patch files are also available on external/avb/contrib/linux/.
The following example shows dm-verity related settings for system-as-root in kernel command line:

```
ro root=/dev/dm-0 rootwait  skip_initramfs init=/init

dm="1 vroot none ro 1,0 5159992 verity 1
PARTUUID=00000016-0000-0000-0000-000000000000
PARTUUID=00000016-0000-0000-0000-000000000000 4096 4096 644999 644999
sha1 d80b4a8be3b58a8ab86fad1b498640892d4843a2
8d08feed2f55c418fb63447fec0d32b1b107e42c 10 restart_on_corruption
ignore_zero_blocks use_fec_from_device
PARTUUID=00000016-0000-0000-0000-000000000000 fec_roots 2 fec_blocks
650080 fec_start 650080"
```

Use device-specific root folders
With system-as-root, after the generic system image (GSI) is flashed on the device (and before running Vendor Test Suite tests), any device-specific root folders added with BOARD_ROOT_EXTRA_FOLDERS are gone because the entire root directory content has been replaced by the system-as-root GSI. The removal of these folders might cause the device to become unbootable if a dependency on the device-specific root folders exists (for example, they're used as mount points).

To avoid this issue, don't use BOARD_ROOT_EXTRA_FOLDERS to add device-specific root folders. If you need to specify device-specific mount points, use /mnt/vendor/<mount point> (added in these changelists). These vendor-specific mount points can be directly specified in both the fstab device tree (for first-stage mount) and the /vendor/etc/fstab.{ro.hardware} file without additional setup (as fs_mgr creates them under /mnt/vendor/* automatically).

This is a **very detailed explanation** of how Android's **partition layout evolved**, especially with **Android 9 and 10**, **system-as-root**, **vendor overlays**, and **dynamic partitions**. Let me simplify and summarize everything step-by-step:

📦 1. What is `system-as-root`?

✅ Traditional Layout (Android 8 and below):

- `boot.img` had:
  - `kernel`
  - `ramdisk` with root directory (`/init`, `/etc`, etc.)
- `system.img` was mounted at `/system`

```pgsql
/ (root)        ← from ramdisk
/system         ← mounted from system.img
/vendor         ← mounted separately
```

✅ System-as-root Layout (Android 9+)

- `boot.img` = Only kernel + small ramdisk (no rootfs)
- `system.img` = Now also holds the **root file system** (`/`).
- So, system.img is **mounted directly as** `/`

```pgsql
/ (root)        ← from system.img
/system         ← subdir inside /
/vendor         ← mounted separately
```

👉 This is **mandatory in Android 10+**
👉 Enables **system-only OTA updates** (i.e., update just system.img)

🗂️ 2. Ramdisk Changes

Version	Where is rootfs?	Ramdisk present?
Android 9	Inside `system.img`	Yes, in boot.img
Android 10+	Inside `system.img`	Yes (required for dm-linear)

You **must**:

- Set `BOARD_BUILD_SYSTEM_ROOT_IMAGE := false`
  (System-as-root is default, so this flag is only for identifying ramdisk use.)

🧰 3. Dynamic Partitions + system-as-root

✅ Dynamic partitions allow:

- Creating/modifying partitions (e.g., system, vendor, product) **without reflashing full images**
- Used in **OTA updates** with A/B support.

```bash
super.img → contains logical partitions: system, product, vendor...
```

→ Dynamic partitions are mounted by **init** in first stage of boot.
→ `dm-linear` maps them like virtual block devices.

🔒 4. dm-verity (Device Mapper + Verified Boot)

dm-verity ensures **no tampering** with system partitions.

Versions:

🔹 **vboot 1.0**

- Kernel parses `dm` parameters from `system.img` metadata.

🔹 **vboot 2.0 (AVB - Android Verified Boot)**

- Bootloader uses `libavb` to get hashtree
- Passes parameters to kernel via command line
- Uses `vbmeta` and `veritykeyid` signatures

🧪 Example kernel cmdline (AVB v2):

```bash
ro root=/dev/dm-0 rootwait skip_initramfs init=/init
dm="system none ro,0 1 android-verity /dev/block/xyz"
veritykeyid=id:abc123
```

🧩 5. Vendor Overlay

Purpose:

Let product partition **override or add files to vendor** at boot (without modifying vendor.img directly).

✅ How it works:

At boot:

- `init` looks inside `/product/vendor_overlay/<version>/`
- If files match `/vendor` dirs and SELinux contexts match → **overlay happens**

Example:

```swift
/product/vendor_overlay/30/lib/libfoo.so → overlays /vendor/lib/libfoo.so
```

⚠️ Limitations:

- **Cannot delete** vendor files
- File context (SELinux label) **must match**

✅ Kernel must support:

```bash
CONFIG_OVERLAY_FS=y
```

🧱 6. Updating Non-A/B Devices to System-as-root

Non-A/B (legacy) devices need:

- Separate `boot` and `recovery` partitions (since no slot switching)
- System.img now must act as rootfs

Update steps:

1. Restructure boot and system partitions
2. Enable dm-verity
3. Avoid using `BOARD_ROOT_EXTRA_FOLDERS`
4. Use `/mnt/vendor/xyz` for device-specific mount points

⛔ Why not use `BOARD_ROOT_EXTRA_FOLDERS`?

- System-as-root **replaces the root directory** with the system image
- Custom folders in root get wiped
- Solution: use `/mnt/vendor/*` for custom mount points

🧭 Summary Table

Feature	Android 8 and below	Android 9+ / 10+
`system-as-root`	Optional / Legacy	✅ Default (mandatory in 10+)
`boot.img` structure	Kernel + full ramdisk	Kernel + minimal ramdisk
Rootfs location	From boot ramdisk	From `system.img`
Dynamic Partitions	❌ Not supported	✅ Supported
OTA Update (system-only)	❌ Risky	✅ Supported with Treble
Vendor overlay	❌ Manual changes needed	✅ Supported via OverlayFS
Device-specific root folders	Used (BOARD_ROOT_EXTRA_*)	❌ Avoid, use `/mnt/vendor`
dm-verity integration	Basic (vboot 1.0)	✅ vboot 2.0 (AVB)

Would you like:

- A visual **diagram** for system-as-root + dynamic partitions?
- Step-by-step **implementation for vendor overlay**?
- Commands to **inspect this on a real device**?

Let me know how you'd like to proceed!

6. what is rootfs explain in simple words with example

Sure! Let's break down **what `rootfs` means** in **very simple words**, with examples 👇

🧾 What is `rootfs`?

`rootfs` = root filesystem = the main folder structure of Linux or Android when it boots up.

When the system starts, the Linux kernel needs to know:

> "Where do I find all the files I need to start the OS?"

That location is called the **root filesystem**, and it's always mounted at `/` **(slash)**.

🏠 Simple Analogy

Imagine a computer's storage like a **house**:

- `/` = **Main entrance**
- `/bin` = tools like screwdriver, hammer
- `/etc` = instruction manuals
- `/system` = house rules
- `/vendor` = devices like air conditioner remote, washing machine manuals
- `/data` = your personal stuff

So, **rootfs** is the **whole house's foundation and entry point**.

🔍 What's inside `rootfs`?

A minimal rootfs usually includes:

```sql
/
├── bin/        → Basic commands like sh, ls
├── sbin/       → System commands (like init)
├── etc/        → Config files
├── init        → First program that runs
├── dev/        → Device files
├── proc/       → Virtual system info
├── sys/        → Kernel interface
├── system/     → Android OS (in Android)
├── vendor/     → HALs and drivers (in Android)
```

📱 Example: Android's rootfs

When an Android phone boots:

- The **Linux kernel** starts
- It mounts the **rootfs** at `/`
- Then it runs `/init` inside the rootfs

In older Android:

- `rootfs` came from the **ramdisk inside `boot.img`**

In newer Android (system-as-root):

- **`rootfs`** is part of the **`system.img`**

So **`/init`**, **`/system`**, **`/vendor`** are **all inside system.img**

📦 Where does rootfs come from?

System	Where rootfs comes from
Traditional Linux	Extracted from **`initrd`** or disk
Android (old)	From **`ramdisk`** inside **`boot.img`**
Android (10+)	From **`system.img`** (system-as-root)

🛠️ Example in Android 10+:

Imagine you mount the system image like this:

```bash
mount system.img /mnt/system
ls /mnt/system
```

You'll see:

```csharp
init
bin/
etc/
system/
vendor/
```

That's your **rootfs**!

✅ Summary

Term	Meaning
`rootfs`	The root filesystem mounted at **`/`**
Contains	**`init`**, **`bin/`**, **`etc/`**, **`system/`** etc.
Needed for	Booting the OS
In Android	Comes from **`system.img`** (new) or **`boot.img`** ramdisk (old)

Would you like a **visual diagram** of this? Or how to **inspect or change rootfs** on an Android device?

Let me know!

7. why move from boot.img to system.img

Excellent question! Let's answer **why Android moved the root filesystem (rootfs) from `boot.img` to `system.img`** — in **simple terms**, with reasons and benefits.

📦 Old Method: `boot.img` Had Rootfs

In Android 8 and below:

- The root filesystem (rootfs) was inside the **`ramdisk` of `boot.img`**.
- This included files like:

```pgsql
/
├── init
├── etc/
├── system/ (mount point only)
├── vendor/ (mount point only)
```

Issues:

- The **rootfs was separate from `system.img`**
- You had to build and flash **both `boot.img` and `system.img`** to make any root-level changes.
- OTA (Over-the-air) updates were **more complex**.

🚀 New Method: Rootfs moved to `system.img` (Android 9+)

Starting Android 9 (mandatory in Android 10+):

- Android switched to **system-as-root**:
  - Rootfs is now inside **`system.img`** itself.
  - **`boot.img`** only has: **kernel + small ramdisk**

🧠 So... Why Move rootfs from `boot.img` to `system.img`?

✅ 1. Simplifies Updates (OTAs)

- With **system-as-root**, Android can update only **`system.img`** without touching **`boot.img`**.
- Easier for **modular OTA updates**.
- Faster and more reliable system updates.

✅ 2. Enables A/B (Seamless) Updates

- A/B updates require that the system is completely self-contained in one partition.
- Putting rootfs inside **`system.img`** makes it **easier to swap slots** (system_a ↔ system_b) without changing boot.img.

✅ 3. Better Separation of Concerns

- **`boot.img`** only contains:
  - Kernel
  - Ramdisk needed for **`dm-verity`** or early init
- All **Android framework files** go to **`system.img`**, where they belong.

✅ 4. Support for Dynamic Partitions

- Android 10+ introduced **dynamic partitions** (super.img).
- These are created and resized at runtime.
- **`boot.img`** can't be dynamic, but **`system.img`** can — so rootfs must go into system.

✅ 5. Consistency with GSIs

- A GSI (Generic System Image) expects a certain rootfs layout.
- System-as-root allows **one universal** **`system.img`** for many devices.

✅ 6. Improved Security

- Rootfs in **`system.img`** is protected by **dm-verity and AVB** (Verified Boot).
- Prevents tampering with root files.
- **`boot.img`** had weaker protection for rootfs content.

🧩 Summary Table

Feature	Old (boot.img rootfs)	New (system.img rootfs)
Location of **/** (rootfs)	In ramdisk of **`boot.img`**	In **`system.img`**
OTA Updates	Complex, multiple images	Easier, system-only
Seamless (A/B) updates	Not fully compatible	Fully compatible
dm-verity + AVB	Partial	Stronger and unified
Dynamic Partitions support	❌ No	✅ Yes

✅ Example Boot Flow (Android 10+):

```text
1. Bootloader loads boot.img → kernel + ramdisk
2. Kernel starts → mounts system.img as rootfs (/)
3. system.img contains /init, /system, /vendor, etc.
4. Android boots from system.img directly
```

Would you like:

- A **boot diagram** comparing old vs. new?
- Commands to inspect rootfs location on a real device?
- A visual example of **`system-as-root`** vs legacy?

Let me know how you'd like to continue!

8. dynamic partition explain in simple words with example

Great! Let's explain **Dynamic Partitions in Android** in **simple words**, with an example and diagrams.

🧱 What is a Partition?

In Android, a **partition** is just a **section of internal storage** reserved for a specific purpose:

Partition	Purpose
`boot`	Kernel + ramdisk
`system`	Android OS files
`vendor`	Device drivers & HALs
`userdata`	Apps, user data, etc.

Earlier, these partitions had **fixed sizes** and were **separate** on flash storage.

🔄 What are Dynamic Partitions?

Dynamic Partitions = virtual partitions created using **logical volumes** on a **shared space** called `super.img`.

Instead of writing each partition (`system`, `vendor`, etc.) as a separate block on storage, Android now:

✅ Combines them into one big partition: `super.img`
✅ Creates **logical partitions** (like containers) inside it
✅ These can be **resized, created, or deleted** without touching actual storage layout

---

📦 Example

🛠️ Traditional way (Fixed partitions):

Storage divided like this:

```lua
|--- boot ---|--- system ---|--- vendor ---|--- product ---|
```

If `system` is full, and `vendor` has extra space → **you can't reuse vendor's space.**

🔄 Dynamic Partitions (Super Partition):

All of these go into one big partition `super.img`, which acts like a container:

```sql
|--------- super.img ---------|
| system | vendor | product   | ← logical partitions
```

Now, you can resize `system` to be bigger and shrink `vendor` if needed.

🧩 Why Dynamic Partitions?

Feature	Benefit
🔁 Resizable	Easily resize `system`, `vendor`, etc. during OTA
⚙️ OTA Friendly	Seamless updates without wiping full device
🔄 Create/Delete	Add/remove partitions as needed
🪛 GSI Support	Generic System Images need flexible layout
📱 Shared Space	Better use of internal storage

✅ Where is it Used?

Dynamic partitions are used in:

- Devices running **Android 10 or later**
- Devices using **seamless (A/B) updates**
- Devices flashing **GSI images**

🧠 What's Inside `super.img`?

Use this command:

```bash
lpdump
```

Sample output:

```sql
super partition layout:
  logical partition: system_a
  logical partition: vendor_a
  logical partition: product_a
```

So all logical partitions live inside one physical `super` block.

💡 Analogy: Bento Box

Imagine storage as a **bento box**:

🍱 `super.img` = the box
🍤 `system`, 🍙 `vendor`, 🥗 `product` = the compartments
With dynamic partitions, you can **resize compartments inside the box** without changing the box itself.

🛠️ How OTA Works with Dynamic Partitions

1. OTA update comes
2. `update_engine` uses `lvm` to resize or create new partitions
3. Only required partitions (e.g., `system_a`) are updated
4. Others stay untouched

This makes **OTA faster and safer**.

🧭 Summary

Term	Meaning
Dynamic Partition	Virtual partition inside `super.img`
super.img	Combined storage space for system, vendor, product
Logical Partition	A resizable partition inside super
Benefits	OTA-friendly, flexible, efficient storage

Would you like:

- A **diagram comparing fixed vs dynamic layout**?
- Sample `super.img` structure dump?

- A command walkthrough on how to list logical partitions on your Android device?

Let me know!

9. which block device super.img uses

Great question!

The `super.img` (dynamic partition container) is backed by a **block device**, typically located under `/dev/block/by-name/` or `/dev/block/platform/.../by-name/`.

📦 The block device for `super.img`

On modern Android devices (with dynamic partitions), the physical storage block that holds `super.img` is usually:

```swift
/dev/block/by-name/super
```

or sometimes:

```swift
/dev/block/platform/soc/.../by-name/super
```

🔹 This `super` block device is where the actual `super.img` lives on flash storage.

🔍 How to check this on a real device:

You can find the super block device using:

```bash
ls -l /dev/block/by-name/super
```

Output might look like:

```javascript
lrwxrwxrwx 1 root root 21 2021-01-01 12:00 super -> /dev/block/sda21
```

👉 This means `/dev/block/sda21` is the actual block device for the `super` partition.

🛠️ Inside the `super` block device

Once mounted, the `super` block contains logical partitions like:

- `system_a`
- `vendor_a`
- `product_a`

These are **virtual (logical) partitions**, created using **dm-linear** and **metadata in super.img**.

📌 Summary

Item	Value
Super block device	`/dev/block/by-name/super`
Underlying physical device	Often `/dev/block/sdaXX`
Contains	Logical partitions like `system_a`, `vendor_a`, etc.
Managed by	`lvm`, `dm-linear`, `fs_mgr`, `update_engine`

Would you like to:

- See how to extract or inspect logical partitions inside `super.img`?
- Understand how `lpdump` or `lpunpack` works?
  Let me know!

10. dm-verity explain in simple words with example

Absolutely! Let's explain `dm-verity` in **simple words**, with a clear example and why it's important in Android.

🔐 What is `dm-verity`?

`dm-verity` stands for **device-mapper verity**.
It's a **Linux kernel feature** that ensures **your device's partitions haven't been tampered with**.

💡 In Simple Terms:

> "`dm-verity` checks if the files in the `system` or `vendor` partition are exactly what they should be — and not secretly modified or hacked."

It's like a **security seal** on your Android OS files.

📦 Where is it used?

`dm-verity` is used to protect **read-only partitions** like:

Partition	Contains
`system`	Android framework, apps
`vendor`	HALs, drivers
`product`	Manufacturer features

🧠 Why we need it?

Without dm-verity:

- Malware or rootkits could change system files silently
- Root access could be gained by modifying `system.img`

With dm-verity:

- Android will **refuse to boot** if the `system.img` is tampered
- Ensures **Verified Boot** is secure and trustworthy

⚙️ How does `dm-verity` work?

1. **Hashes are calculated** for every block (small chunk) in the system image when the firmware is built.
2. A **Merkle tree** (a hash tree) is created — root hash is saved.
3. At boot time:
    - As the OS reads any block from system.img
    - `dm-verity` checks its hash against the tree
    - If any mismatch is found → 🔥 Android **refuses to boot**

🧩 Think of this like a big box of LEGO:

- You build a tree of small checksums (block hashes) to verify the structure.
- If someone changes even one brick (file), the whole tree says "❌ invalid!"

🧪 Real-World Example (Android Boot)

When booting, your **kernel command line** may include:

```bash
dm="system none ro,0 1 android-verity /dev/block/sdaX"
```

This tells the kernel:

> "Mount `/system` as read-only, and verify it using dm-verity hashes."

🔧 Related files

File	Purpose
`verity metadata`	Contains hash tree & root hash
`vbmeta.img`	Used by AVB to sign & verify
`init` & `fstab`	Set up dm-verity mounts

📌 Summary

Term	Meaning
`dm-verity`	Verifies read-only partitions block-by-block
Protects	`system`, `vendor`, `product`, etc.
Uses	Merkle hash trees to detect tampering
Prevents	Malware, rootkits, unauthorized file changes
Part of	Android Verified Boot (AVB)

✅ Visual Summary

```bash
              Read /system/lib/libc.so
                        ↓
        dm-verity checks hash of block
                        ↓
          Matches hash in Merkle tree?
                ↙               ↘
             YES                NO
           Allow read      Panic! (Stop boot)
```

Would you like:

- A **Merkle tree diagram**?
- Steps to disable `dm-verity` on a test device (for research)?
- Kernel-side or fstab-level configuration of `dm-verity`?

Let me know how deep you'd like to go!

11. dm-verity vs avb

Great question! Let's clearly explain the difference between **dm-verity** and **AVB (Android Verified Boot)** in **simple words**, with a comparison table and examples.

📦 First, What Are They?

Term	Full Form	Purpose
dm-verity	device-mapper verity	Verifies **data blocks** on partitions at runtime
AVB	Android Verified Boot	Verifies **partition images** before booting

---

🧠 Simple Explanation

🛡️ dm-verity → "Check each block at runtime"

- When Android tries to **read a file** (e.g. `/system/lib/libc.so`),
- `dm-verity` checks if the **block** on disk matches a pre-calculated hash.
- If it doesn't match → 🧨 **Boot fails (or panic).**

🔐 AVB → "Check the image before booting"

- AVB checks **entire partitions** like `boot`, `system`, `vendor`, etc. **before booting**.
- It uses **cryptographic signatures** stored in `vbmeta.img`.
- If image tampered or unsigned → 🚫 **Device won't boot.**

🔧 How They Work Together

✅ **AVB includes dm-verity hash trees** as part of the verification process.

✅ So in Android 10+:

- AVB checks partition signatures → if valid
- Then enables dm-verity to check blocks at runtime

📊 Comparison Table

Feature	`dm-verity`	AVB (Android Verified Boot)
Checks when?	At runtime (when data is read)	At boot (before loading kernel/partitions)
Checks what?	Individual blocks inside partitions	Entire partition images & their signatures
Protects against?	Runtime file tampering, silent changes	Fake or modified images, rootkits
Uses signature?	❌ No (uses hash trees only)	✅ Yes (digital signature with public key)
Depends on?	Merkle hash tree stored in partition	`vbmeta.img`, public key in bootloader
Introduced in Android	Android 4.4 (KitKat)	Android 7.0 (Nexus), mandatory in 8.0+ (AVB 1.0), improved in 10+ (AVB 2.0)
Boot stops if fails?	✅ Yes	✅ Yes

📱 Real Example (Android boot flow with both):

```text
Bootloader:
  1. Loads vbmeta.img
  2. Verifies vbmeta signature (AVB check)
```

```
   3. If valid, loads and verifies boot/system/vendor using AVB
      → AVB passes
   4. Kernel boots with dm-verity enabled
      → Every read from /system is now verified block-by-block
```

🧩 Analogy: Book & Seal

- **AVB** = Sealed box that guarantees "This book hasn’t been swapped"
- **dm-verity** = Each page has a watermark → if someone edits 1 page → caught while reading!

✅ Summary

- **AVB = Signature check at boot (image-level)**
- **dm-verity = Block check during use (runtime-level)**

Together, they ensure:

- You **boot only trusted images**
- And you **don’t read tampered content** at runtime

Would you like:

- A diagram of how AVB + dm-verity work in sequence?
- Commands to inspect or disable AVB/dm-verity (on test devices)?
- Examples of tampering detection?

Let me know!

12. dm-linear

Great question again! Let's explain `dm-linear` in **simple words**, especially in the context of Android.

📦 What is `dm-linear`?

`dm-linear` stands for **device-mapper linear mapping**.
It is a **Linux kernel module** used to **combine parts of physical storage into one virtual block device**.

🧠 In Simple Words:

> "`dm-linear` lets you create a *virtual block device* by stitching together chunks from one or more physical block devices."

You define:

- Start offset on a physical device
- Length of blocks
- And create a new device that maps to them linearly

✅ Android Usage

In Android, `dm-linear` **is used to implement dynamic partitions**.

- Android has a big `super` partition (e.g., `/dev/block/by-name/super`)
- Inside it, it creates virtual partitions like `system_a`, `vendor_a`, etc.
- These logical partitions are **dm-linear mappings** to regions inside `super.img`

🛠️ Example

Let's say:

Physical Block Device	Offset (start)	Length (sectors)	Purpose
/dev/block/sda21	1 MB	512 MB	system_a
/dev/block/sda21	513 MB	256 MB	vendor_a

`dm-linear` will create two virtual devices:

- `/dev/block/dm-0` → maps to `system_a`
- `/dev/block/dm-1` → maps to `vendor_a`

These appear like **real partitions** to Android and mount like:

```bash
/dev/block/dm-0  →  /system
/dev/block/dm-1  →  /vendor
```

📊 How `dm-linear` fits in

Component	Role
`super.img`	Physical block device holding all data
`metadata`	Describes where partitions start/stop
`dm-linear`	Maps logical partitions to those regions
`fstab` & `init`	Use the mapped devices like real partitions

🧩 Analogy: Bookshelf

Imagine a large bookshelf (`super.img`). You want to treat:

- Row 1–10 as the **System Partition**
- Row 11–15 as the **Vendor Partition**

`dm-linear` is like **labeling specific shelf ranges** and pretending each is its own shelf.

🔍 How to See It on Android

Use the command:

```bash
ls -l /dev/block/mapper
```

You'll see something like:

```bash
system_a → /dev/dm-0
vendor_a → /dev/dm-1
```

And then:

```bash
cat /sys/block/dm-0/uevent
```

To see where it's actually mapped from.

🔐 Bonus: `dm-verity + dm-linear`

- `dm-linear` maps the partition (e.g., `/system`) inside super.img
- `dm-verity` verifies it block-by-block for tampering

These two often **work together** on Android for dynamic partitions.

✅ Summary

Term	Meaning
dm-linear	Maps virtual partitions from a big physical device (`super.img`)
Used for	Dynamic partitions (system, vendor, product, etc.)
Works with	`dm-verity` and AVB to secure boot and runtime integrity
Appears as	`/dev/dm-*` devices (e.g., `/dev/block/dm-0`)

Would you like:

- A visual diagram showing `super.img → dm-linear → system_a`?
- Sample metadata that defines a logical partition?
  Let me know!

13. lvm, fs_mgr, and update engine

Let's explain **LVM**, **fs_mgr**, and **update_engine** in **simple, Android-specific terms** — how they work and how they connect, especially with **dynamic partitions** and **OTA updates**.

🔧 1. LVM (Logical Volume Manager)

📦 What it is:

LVM is a Linux kernel feature that lets you treat your storage like **Lego blocks** — you can combine, resize, and split them **without formatting or rebooting**.

📱 In Android:

On Android (especially with dynamic partitions), LVM is used **internally by `fs_mgr` and `update_engine`** via `dm-linear` to:

- Create virtual partitions inside `super.img`
- Resize partitions like `system`, `vendor`, `product`
- Enable **OTA (Over-The-Air) updates** to dynamically modify partition layouts

🧠 Analogy:

> Imagine one big box (`super.img`) and you're dividing it into virtual rooms (`system`, `vendor`) — LVM lets you draw/change room walls anytime.

🗂️ 2. fs_mgr (File System Manager)

📦 What it is:

`fs_mgr` is a component in **Android's init system** (`init.rc`) that:

- **Mounts partitions** during boot
- **Reads `fstab`** (file system table) to know what to mount, where, and how
- Works with `dm-verity`, `dm-linear`, and `AVB`

📱 In Android:

- Uses **LVM/dm-linear** mappings to mount `system`, `vendor`, etc.
- Verifies images via **dm-verity**
- Ensures partitions are mounted **securely** and correctly

🔍 Example:

```bash
/dev/block/mapper/system_a /system ext4 ro,barrier=1 wait,avb
```

This tells `fs_mgr`:

> Mount logical partition `system_a` as `/system` using ext4 and AVB verification.

🔁 3. update_engine

📦 What it is:

`update_engine` is the **OTA updater daemon** that applies Android system updates **in the background**.

📱 In Android:

- Works on **A/B (Seamless) update devices**
- Updates the **inactive slot** (e.g., `system_b`)
- Can **resize partitions** inside `super.img` using LVM + metadata
- Uses `delta payloads` — only changes that are needed

🧠 Analogy:

> It's like a software robot that quietly renovates your second room while you're using the first — then swaps them at reboot.

🔄 How They Work Together (OTA Flow)

```text
              [ Android Update Begins ]
                          ↓
              update_engine downloads OTA
                          ↓
        Parses super metadata → resize logical partitions
                          ↓
              Uses LVM/dm-linear to create new mappings
                          ↓
         Writes new data to `system_b`, `vendor_b`, etc.
                          ↓
         fs_mgr mounts new slot at next reboot (A/B switch)
                          ↓
                  New Android boots ✅
```

✅ Summary

Component	Role in Android
LVM	Manages logical partitions inside `super.img`
fs_mgr	Mounts partitions during boot, handles AVB & verity
update_engine	Applies OTA updates, modifies logical partitions dynamically

---

📦 Bonus Tip: Commands

- List logical partitions:

  ```bash
  lpdump
  ```

- Check mounts and fs_mgr output:

  ```bash
  dmesg | grep fs_mgr
  ```

- OTA logs:

  ```bash
  logcat | grep update_engine
  ```

Would you like:

- A diagram showing this full OTA flow with LVM, fs_mgr, update_engine?
- Sample `fstab` entries and `super.img` metadata?
  Let me know how deep you want to go!

14. merkle tree

Let’s break down the **Merkle Tree** in **simple words**, with an example — especially how it's used in Android (with **`dm-verity`**, **`AVB`**, etc.).

🌳 What is a Merkle Tree?

A **Merkle Tree** is a **tree of hashes**.
It's used to **quickly verify data integrity** — like making sure **nothing is changed or tampered**.

🧠 Simple Explanation:

> Instead of checking every file or block one by one, you check just **one final hash** — the **root hash**.
> If anything was changed in the data, that root hash will no longer match.

🧩 Structure of a Merkle Tree

Here’s a simple example:

```mathematica
         Root Hash
           /   \
     Hash A     Hash B
     /   \       /   \
  D1     D2    D3    D4
```

- **`D1`**, **`D2`**, **`D3`**, **`D4`** = raw data blocks (e.g., from system.img)
- Hashes are computed like:

```mathematica
Hash A = hash(D1 + D2)
Hash B = hash(D3 + D4)
Root Hash = hash(Hash A + Hash B)
```

So, only the **root hash** is needed to verify all data!

🔐 Why is it used?

✅ Fast verification

You don’t need to read the entire file or image — just a small set of hashes.

✅ Tamper detection

If **even 1 bit changes**, it will **break the entire chain**, and the root hash won’t match.

📱 In Android (dm-verity / AVB)

Android builds a Merkle tree for **`system`**, **`vendor`**, **`product`** partitions.

At build time:

- A Merkle tree is generated
- The **root hash** is saved in **`vbmeta.img`**

At boot time:

- `dm-verity` or AVB uses the root hash to verify every block read
- If anything is changed → ❌ boot fails

📦 Example (Android system.img Merkle Tree)

Let’s say system.img is 4 blocks:

```
scss

Block 1 → hash1
Block 2 → hash2
Block 3 → hash3
Block 4 → hash4
```

Then:

```
mathematica

Hash A = hash(hash1 + hash2)
Hash B = hash(hash3 + hash4)
Root  = hash(Hash A + Hash B)
```

This **root hash** is stored in `vbmeta.img`.
At runtime, Android uses `dm-verity` to verify each block against the Merkle tree.

📊 Summary Table

Term	Meaning
Merkle Tree	Tree structure made of hashes
Used for	Verifying data integrity efficiently
In Android	Used in **`dm-verity`**, **AVB**
Root Hash	One final hash representing full data
Change Detection	Any change in data → root hash fails

🎓 Analogy

Imagine a **pyramid of signatures**:

- Every file (leaf) has a signature
- You group them and sign the group
- Then sign the top group again

If someone **fakes 1 file**, the final signature **breaks** — and you know instantly.

Would you like:

- A diagram showing the full Merkle Tree in Android?
- Code to generate a Merkle tree in Python?
  Let me know!

15. Vendor boot partitions

bookmark_border
Android 11 introduced the concept of the Generic Kernel Image (GKI). To enable booting an arbitrary device with the GKI, Android 11 devices can use boot image header version 3. In version 3, all vendor-specific information is factored out of the boot partition and relocated into a new vendor_boot partition. An ARM64 device launching with Android 11 on the 5.4 Linux kernel must support the vendor_boot partition and the updated boot partition format to pass testing with the GKI.

Android 12 devices can use boot image header version 4, which supports including multiple vendor ramdisks in the vendor_boot partition. Multiple vendor ramdisk fragments are concatenated one after another in the vendor ramdisk section. A vendor ramdisk table is used to describe the layout of the vendor ramdisk section and the metadata of each vendor ramdisk fragment.

Partition structure
The vendor boot partition is A/B-ed with virtual A/B and protected by Android Verified Boot.

Version 3
The partition consists of a header, the vendor ramdisk, and the device tree blob (DTB).

Section	Number of pages
Vendor boot header (n pages)	n = (2112 + page_size - 1) / page_size
Vendor ramdisk (o pages)	o = (vendor_ramdisk_size + page_size - 1) / page_size
DTB (p pages)	p = (dtb_size + page_size - 1) / page_size

Version 4
The partition consists of a header, the vendor ramdisk section (consisting of all vendor ramdisk fragments, concatenated), the device tree blob (DTB), and the vendor ramdisk table.

Section	Number of pages
Vendor boot header (n pages)	n = (2128 + page_size - 1) / page_size
Vendor ramdisk fragments (o pages)	o = (vendor_ramdisk_size + page_size - 1) / page_size
DTB (p pages)	p = (dtb_size + page_size - 1) / page_size
Vendor ramdisk table (q pages)	q = (vendor_ramdisk_table_size + page_size - 1) / page_size
Bootconfig (r pages)	r = (bootconfig_size + page_size - 1) / page_size

Note: For details on the vendor ramdisk and the handling of kernel modules, see Kernel Module Support.
Vendor boot header
The contents of the vendor boot partition header consist primarily of data that has been relocated there from the boot image header. It also contains information about the vendor ramdisk.

Version 3

```
struct vendor_boot_img_hdr_v3
{
#define VENDOR_BOOT_MAGIC_SIZE 8
    uint8_t magic[VENDOR_BOOT_MAGIC_SIZE];
    uint32_t header_version;
    uint32_t page_size;           /* flash page size we assume */

    uint32_t kernel_addr;         /* physical load addr */
    uint32_t ramdisk_addr;        /* physical load addr */

    uint32_t vendor_ramdisk_size; /* size in bytes */

#define VENDOR_BOOT_ARGS_SIZE 2048
    uint8_t cmdline[VENDOR_BOOT_ARGS_SIZE];

    uint32_t tags_addr;           /* physical addr for kernel tags */

#define VENDOR_BOOT_NAME_SIZE 16
    uint8_t name[VENDOR_BOOT_NAME_SIZE]; /* asciiz product name */
    uint32_t header_size;         /* size of vendor boot image header in
                                   * bytes */
    uint32_t dtb_size;            /* size of dtb image */
    uint64_t dtb_addr;            /* physical load address */
};
```

Version 4

```
struct vendor_boot_img_hdr_v4
{
#define VENDOR_BOOT_MAGIC_SIZE 8
    uint8_t magic[VENDOR_BOOT_MAGIC_SIZE];
    uint32_t header_version;
    uint32_t page_size;           /* flash page size we assume */

    uint32_t kernel_addr;         /* physical load addr */
    uint32_t ramdisk_addr;        /* physical load addr */

    uint32_t vendor_ramdisk_size; /* size in bytes */

#define VENDOR_BOOT_ARGS_SIZE 2048
    uint8_t cmdline[VENDOR_BOOT_ARGS_SIZE];

    uint32_t tags_addr;           /* physical addr for kernel tags */

#define VENDOR_BOOT_NAME_SIZE 16
    uint8_t name[VENDOR_BOOT_NAME_SIZE]; /* asciiz product name */
    uint32_t header_size;         /* size of vendor boot image header in
                                   * bytes */
    uint32_t dtb_size;            /* size of dtb image */
    uint64_t dtb_addr;            /* physical load address */

    uint32_t vendor_ramdisk_table_size; /* size in bytes for the vendor ramdisk table */
    uint32_t vendor_ramdisk_table_entry_num; /* number of entries in the vendor ramdisk table */
    uint32_t vendor_ramdisk_table_entry_size; /* size in bytes for a vendor ramdisk table entry */
    uint32_t bootconfig_size; /* size in bytes for the bootconfig section */
};

#define VENDOR_RAMDISK_TYPE_NONE 0
#define VENDOR_RAMDISK_TYPE_PLATFORM 1
#define VENDOR_RAMDISK_TYPE_RECOVERY 2
#define VENDOR_RAMDISK_TYPE_DLKM 3

struct vendor_ramdisk_table_entry_v4
{
    uint32_t ramdisk_size; /* size in bytes for the ramdisk image */
    uint32_t ramdisk_offset; /* offset to the ramdisk image in vendor ramdisk section */
    uint32_t ramdisk_type; /* type of the ramdisk */
#define VENDOR_RAMDISK_NAME_SIZE 32
    uint8_t ramdisk_name[VENDOR_RAMDISK_NAME_SIZE]; /* asciiz ramdisk name */

#define VENDOR_RAMDISK_TABLE_ENTRY_BOARD_ID_SIZE 16
    // Hardware identifiers describing the board, soc or platform which this
    // ramdisk is intended to be loaded on.
    uint32_t board_id[VENDOR_RAMDISK_TABLE_ENTRY_BOARD_ID_SIZE];
};
```

vendor_ramdisk_size is the total size of all the vendor ramdisk fragments.
ramdisk_type denotes the type of the ramdisk, possible values are:
VENDOR_RAMDISK_TYPE_NONE indicates the value is unspecified.
VENDOR_RAMDISK_TYPE_PLATFORM ramdisks contain platform specific bits. The bootloader must always load these into memory.
VENDOR_RAMDISK_TYPE_RECOVERY ramdisks contain recovery resources. The bootloader must load these into memory when booting into recovery.
VENDOR_RAMDISK_TYPE_DLKM ramdisks contain dynamic loadable kernel modules.
ramdisk_name is an unique name of the ramdisk.
board_id is a vector of vendor defined hardware identifiers.
Note: To view AOSP details on all supported boot and vendor boot image headers, refer to system/tools/mkbootimg/include/bootimg/bootimg.h.
Bootloader support
Because the vendor boot partition contains information (such as flash page size, kernel, ramdisk load addresses, the DTB itself) that previously existed in the boot partition, the bootloader must access both the boot and vendor boot partitions to have enough data to complete booting.

The bootloader must load the generic ramdisk into memory immediately following the vendor ramdisk (the CPIO, Gzip, and lz4 formats support this type of concatenation). Don't page align the generic ramdisk image or introduce any other space between it and the end of the vendor ramdisk in memory. After the kernel decompresses, it extracts the concatenated file into an initramfs, which results in a file structure that's a generic ramdisk overlaid on the vendor ramdisk file structure.

Because the generic ramdisk and vendor ramdisk get concatenated, they must be in the same format. The GKI boot image uses an lz4-compressed generic ramdisk, so a device that is GKI-compliant must use an lz4-compressed vendor ramdisk. The configuration for this is shown below.

The bootloader requirements for supporting bootconfig are explained in Implement Bootconfig.

Multiple vendor ramdisks (version 4)
With boot image header version 4, the bootloader can select either a subset or all of the vendor ramdisks to load as the initramfs during boot time. The vendor ramdisk table contains the metadata of each ramdisk, and can aid the bootloader in deciding which ramdisks to load. The bootloader can decide the order to load the selected vendor ramdisks, as long as the generic ramdisk is loaded last.

For example, the bootloader can omit loading vendor ramdisks of type VENDOR_RAMDISK_TYPE_RECOVERY during normal boot to conserve resources, so only vendor ramdisks of type VENDOR_RAMDISK_TYPE_PLATFORM and VENDOR_RAMDISK_TYPE_DLKM are loaded into memory. On the other hand, vendor ramdisks of type VENDOR_RAMDISK_TYPE_PLATFORM, VENDOR_RAMDISK_TYPE_RECOVERY and VENDOR_RAMDISK_TYPE_DLKM are loaded into memory when booting into recovery mode.

Alternatively, the bootloader can ignore the vendor ramdisk table and load the entire vendor ramdisk section. This has the same effect as does loading all of the vendor ramdisk fragments in the vendor_boot partition.

Build support
To implement vendor boot support for a device:

Set BOARD_BOOT_HEADER_VERSION to 3 or greater.

Set BOARD_RAMDISK_USE_LZ4 to true if your device is GKI-compliant, or if it otherwise uses an lz4-compressed generic ramdisk.

Set BOARD_VENDOR_BOOTIMAGE_PARTITION_SIZE to an appropriate size for your device, considering the kernel modules that must go on the vendor ramdisk.

Update AB_OTA_PARTITIONS to include vendor_boot and any vendor-specific lists of OTA partitions on the device.

Copy your device fstab into /first_stage_ramdisk in the vendor_boot partition, not the boot partition. For example, $(LOCAL_PATH)/fstab.hardware:$(TARGET_COPY_OUT_VENDOR_RAMDISK)/first_stage_ramdisk/fstab.$(PRODUCT_PLATFORM).

To include multiple vendor ramdisks in vendor_boot:

Set BOARD_BOOT_HEADER_VERSION to 4.
Set BOARD_VENDOR_RAMDISK_FRAGMENTS to a list of logical vendor ramdisk fragment names to be included in vendor_boot.

Important: For Android 12 and higher, usage of the name default is reserved. Giving the name default to vendor ramdisk fragments isn't allowed under any circumstances. To learn more, see the Moving Fastboot to Userspace page, under mkbootimg.
To add a prebuilt vendor ramdisk, set BOARD_VENDOR_RAMDISK_FRAGMENT.$(vendor_ramdisk).PREBUILT to the prebuilt path.

To add a DLKM vendor ramdisk, set BOARD_VENDOR_RAMDISK_FRAGMENT.$(vendor_ramdisk).KERNEL_MODULE_DIRS to the list of kernel module directories to be included.

Set BOARD_VENDOR_RAMDISK_FRAGMENT.$(vendor_ramdisk).MKBOOTIMG_ARGS to mkbootimg arguments. These are the --board_id[0-15] and --ramdisk_type arguments for the vendor ramdisk fragment. For DLKM vendor ramdisk, the default --ramdisk_type would be DLKM if it's not otherwise specified.

To build recovery resources as a standalone recovery ramdisk in vendor_boot:

Set BOARD_BOOT_HEADER_VERSION to 4.
Set BOARD_MOVE_RECOVERY_RESOURCES_TO_VENDOR_BOOT to true.
Set BOARD_INCLUDE_RECOVERY_RAMDISK_IN_VENDOR_BOOT to true.
This adds a vendor ramdisk fragment whose ramdisk_name is recovery and ramdisk_type is VENDOR_RAMDISK_TYPE_RECOVERY. The ramdisk then contains all recovery files, which are files installed under $(TARGET_RECOVERY_ROOT_OUT).
mkbootimg arguments

Argument	Description
--ramdisk_type	The type of the ramdisk, can be one of NONE, PLATFORM, RECOVERY or DLKM.
--board_id[0-15]	Specify the board_id vector, defaults to 0.

Following is an example configuration:

BOARD_KERNEL_MODULE_DIRS := foo bar baz
BOARD_BOOT_HEADER_VERSION := 4
BOARD_VENDOR_RAMDISK_FRAGMENTS := dlkm_foobar
BOARD_VENDOR_RAMDISK_FRAGMENT.dlkm_foobar.KERNEL_MODULE_DIRS := foo bar
BOARD_VENDOR_RAMDISK_FRAGMENT.dlkm_foobar.MKBOOTIMG_ARGS := --board_id0 0xF00BA5 --board_id1 0xC0FFEE
The resulting vendor_boot would contain two vendor ramdisk fragments. The first one is the "default" ramdisk, which contains the DLKM directory baz and the rest of the files in $(TARGET_VENDOR_RAMDISK_OUT). The second one is the dlkm_foobar ramdisk, which contains the DLKM directories foo and bar, and the --ramdisk_type defaults to DLKM.

Here's a **simple explanation** of the **`vendor_boot` partition** in Android (added in Android 11+) and why it exists, especially in the context of **GKI (Generic Kernel Image)** and **modular kernel booting**.

🧩 Why `vendor_boot`?

In **Android 11**, Google introduced **GKI** — a *universal boot kernel* that works across many devices.

To make this work:

- **Generic parts** (common to all devices) go in **`boot.img`**
- **Device/vendor-specific parts** go in a new partition called **`vendor_boot.img`**

So instead of putting everything in **`boot.img`**, Android split it into:

```sql
boot.img        → GKI kernel + generic ramdisk
vendor_boot.img → device-specific ramdisk + DTBs + kernel modules
```

🏗️ Boot Partition Structure

`boot.img` (GKI-compliant):

- Contains only:
  - Kernel (common for all devices)
  - **Generic ramdisk** (minimal init files, same for everyone)

`vendor_boot.img`:

- Contains:
  - **Vendor ramdisk(s)** (device-specific init scripts, fstab, etc.)
  - **DTBs** (Device Tree Blobs — hardware descriptions)
  - **Vendor kernel modules** (DLKMs)
  - **Bootconfig** (kernel boot args)

🧠 Why Split?

This separation allows:
✅ One universal kernel (**`boot.img`**)
✅ Device vendors to customize only **`vendor_boot.img`**
✅ Smaller OTA updates (you only need to update what changed)

🛠️ Header Versions

Version	Introduced in	Features
v3	Android 11	Basic **`vendor_boot`**, one vendor ramdisk

Version	Introduced in	Features
v4	Android 12	Multiple vendor ramdisks, metadata table

🗂️ What's in `vendor_boot.img` v4?

```
plaintext

[Header] → metadata (page size, sizes, addresses)
↓
[Vendor Ramdisks] → one or more compressed fragments
↓
[Device Tree Blob (DTB)] → hardware info
↓
[Vendor Ramdisk Table] → describes each fragment
↓
[Bootconfig] → kernel arguments
```

Each **vendor ramdisk fragment** has:

- A name
- A type (PLATFORM, RECOVERY, DLKM)
- Optional board_id (for device matching)

🧪 Types of Vendor Ramdisks

Type	Use Case
`PLATFORM`	Boot-time init files, fstab, vendor props
`RECOVERY`	Used only when booting into recovery mode
`DLKM` (kernel modules)	Dynamic Loadable Kernel Modules
`NONE`	No specific type (default)

🖼️ Boot Flow (Simplified)

```
text

[Bootloader]
   ↓
Load vendor_boot.img:
   - All vendor ramdisks (based on mode: normal or recovery)
   - DTBs
   - Kernel modules
   ↓
Load boot.img:
   - Kernel
   - Generic ramdisk
   ↓
[Kernel decompresses both ramdisks → creates one merged initramfs]
   ↓
/init launches Android
```

🛠️ Build System Config Example

```
makefile

# Android.mk or BoardConfig.mk
BOARD_BOOT_HEADER_VERSION := 4
BOARD_VENDOR_RAMDISK_FRAGMENTS := dlkm_foobar
BOARD_VENDOR_RAMDISK_FRAGMENT.dlkm_foobar.KERNEL_MODULE_DIRS := foo bar
BOARD_VENDOR_RAMDISK_FRAGMENT.dlkm_foobar.MKBOOTIMG_ARGS := \
    --ramdisk_type DLKM --board_id0 0xF00BA5 --board_id1 0xC0FFEE
```

🧪 Important Notes

- **`vendor_boot.img`** is **A/B partitioned** (has **`_a`**, **`_b`** slots).

- **lz4 compression** is required if using GKI.
- The **bootloader must concatenate vendor + generic ramdisks** properly.
- **fstab must now be placed in `vendor_boot/first_stage_ramdisk`**.

✅ Summary

Feature	boot.img	vendor_boot.img
Kernel	✅ (GKI kernel)	❌
Generic Ramdisk	✅	❌
Vendor Ramdisk(s)	❌	✅ (Platform, DLKM, etc.)
Device Tree (DTB)	❌	✅
Kernel Modules (DLKM)	❌	✅
Bootconfig	❌	✅

Would you like:

- A **visual diagram** of boot + vendor_boot structure?
- An **example mkbootimg command**?
- A **script to extract vendor_boot.img** contents?

Let me know!

16. ⤷
\[Bootloader\] ↓ Load vendor_boot.img: - All vendor ramdisks (based on mode: normal or recovery) - DTBs - Kernel modules ↓ Load boot.img: - Kernel - Generic ramdisk ↓ \[Kernel decompresses both ramdisks → creates one merged initramfs\] ↓ /init launches Android
why vendor_boot.img first then boot.img here

Great question!

✅ Why does the bootloader load `**vendor_boot.img first**`, then `boot.img`?

The answer lies in how Android's **GKI (Generic Kernel Image)** booting process works, especially from **Android 11** onward with **boot header v3+**.

🧠 TL;DR (Simple Answer):

The **vendor-specific information** (like kernel modules, fstab, DTB, etc.) in **`vendor_boot.img` must be loaded into memory *before*** the kernel from **`boot.img`** starts running — because the kernel needs them **during its very early boot stage.**

🧩 Detailed Explanation:

Boot Component	What it Contains (GKI model)	Why Load First?
`vendor_boot.img`	- **Vendor ramdisks** (e.g., **`/first_stage_ramdisk/fstab`**) - **DTBs** (Device Tree Blobs) - **Vendor kernel modules (DLKM)**	These are **required by the kernel very early**, during **first-stage init** (before even **`init.rc`**)
`boot.img`	- **Kernel (GKI)** - **Generic ramdisk**	The kernel is loaded *after* the vendor bits are prepared

---

🔁 Boot Sequence Summary

```
text

1. Bootloader loads vendor_boot.img first:
   - Loads DTB (hardware info)
   - Loads vendor ramdisk(s)
   - Prepares first-stage init files (fstab, kernel modules)

2. Then loads boot.img:
   - Loads the kernel (GKI)
   - Loads the generic ramdisk
   - Concatenates vendor + generic ramdisk (creates initramfs)

3. Kernel decompresses and overlays both ramdisks

4. /init from the merged ramdisk is executed → Android starts
```

📌 Real Reason Why Order Matters

- **`fstab`** is in **`vendor_boot.img`** and used in **first stage init**.
- **DTB** is hardware-specific and tells the kernel how to talk to the device.
- **DLKMs** (kernel modules) need to be loaded early for hardware support.

If you **load `boot.img` first**, the kernel won't have access to these vendor-specific bits — so the system might **crash or fail to mount partitions like `/vendor`, `/odm`, etc.**

🔄 Analogy

Imagine booting Android like cooking a meal:

- **`vendor_boot.img`** = the **kitchen** and **ingredients** (vendor data, hardware config, modules)
- **`boot.img`** = the **chef** and **recipe** (kernel and init system)

You must set up the kitchen before the chef can start cooking.

✅ Conclusion

> The bootloader **must load `vendor_boot.img` first** so the kernel has all the hardware-specific and initialization data **ready in memory** when it starts. Only then does it load `boot.img` with the generic kernel and ramdisk.

Let me know if you want:

- A visual diagram of this flow
- A comparison with pre-GKI booting
- Or a simple script to dump contents of both images

17. Vendor and ODM DLKM partitions

bookmark_border
Android 11 introduces the concept of the Generic Kernel Image and the vendor boot partition. The vendor boot partition stores kernel modules compatible with GKI, and is loaded by first stage init. Kernel Modules prior to Android 11's release are also stored in vendor and ODM partitions, and are loaded by vendor processes.

For Android 11 or higher, the kernel and all kernel modules can be updated independently from the rest of the partitions. To enable updates for kernel modules stored in the vendor partition (without a vendor partition update), move all vendor partition modules to a new partition called Vendor DLKM (dynamically loadable kernel module). You can then update this partition independently. Similarly, you can move all kernel modules stored in the ODM partition to a new partition called ODM DLKM. This partition can be updated independently as well.

Partition location
vendor_dlkm and odm_dlkm partitions are located in the super partition as another dynamic partition.

vendor_dlkm contents in /vendor/lib/modules
Vendor kernel modules
modprobe config files
A modules.load file
odm_dlkm contents in /odm/lib/modules
ODM kernel modules
modprobe config files
A modules.load file
See Kernel Module Support for further details on kernel module config files.

Build support
Building vendor_dlkm and odm_dlkm is a similar process to building other dynamic partitions.

vendor_dlkm build example
Build vendor_dlkm as shown in the following examples.

BoardConfig.mk

```
BOARD_USES_VENDOR_DLKMIMAGE := true
BOARD_VENDOR_DLKMIMAGE_FILE_SYSTEM_TYPE := ext4
TARGET_COPY_OUT_VENDOR_DLKM := vendor_dlkm
BOARD_<GROUP_NAME>_PARTITION_LIST += vendor_dlkm
```

Replace <GROUP_NAME> with the appropriate name of the update group. The update group should be the group that the vendor partition is in.

For A/B and Virtual A/B devices, device.mk

```
AB_OTA_PARTITIONS += vendor_dlkm
```

fstab

Add the following entry for vendor_dlkm to fstab. Change flags according to the device. Use the CL Add vendor_dlkm to CF as an example.

```
vendor_dlkm /vendor_dlkm ext4 noatime,ro,errors=panic wait,logical,first_stage_mount,slotselect,avb
```

odm_dlkm build example
Build odm_dlkm as shown in the following examples.

BoardConfig.mk

```
BOARD_USES_ODM_DLKIMAGE := true
BOARD_ODM_DLKIMAGE_FILE_SYSTEM_TYPE := ext4
TARGET_COPY_OUT_ODM_DLKM := odm_dlkm
BOARD_<group_name>_PARTITION_LIST += odm_dlkm
```

For A/B and Virtual A/B devices, device.mk

AB_OTA_PARTITIONS += odm_dlkm
fstab

Add the following entry for odm_dlkm to fstab. Change flags according to the device. Use the CL Add odm_dlkm to CF as an example.

odm_dlkm /odm_dlkm ext4 noatime,ro,errors=panic wait,logical,first_stage_mount,slotselect,avb
Copy kernel modules into a partition
To select the kernel modules you want to copy into the vendor_dlkm partition, list them in BOARD_VENDOR_KERNEL_MODULES.

If you want to override the contents of modules.load, you can specify it in BOARD_VENDOR_KERNEL_MODULES_LOAD.

At build time, modules listed in BOARD_VENDOR_KERNEL_MODULES are installed in $ANDROID_PRODUCT_OUT/vendor_dlkm/lib/modules. A symbolic link is created at /vendor/lib/modules that leads to /vendor_dlkm/lib/modules.

Similarly, to select the kernel modules you want to copy into the odm_dlkm partition, list them in BOARD_ODM_KERNEL_MODULES. The platform build runs depmod on the modules and copies the depmod output files into the image. The build creates a modules.load file and stores it in the image. This file contains all of the modules listed in BOARD_ODM_KERNEL_MODULES.

If you want to override the contents of modules.load, you can specify it in BOARD_ODM_KERNEL_MODULES_LOAD.

At build time, modules listed in BOARD_ODM_KERNEL_MODULES is installed in $ANDROID_PRODUCT_OUT/odm_dlkm/lib/modules. A symbolic link is created at /odm/lib/modules that leads to /odm_dlkm/lib/modules.

Always use /vendor/lib/modules and /odm/lib/modules for vendor and ODM kernel modules.

Never use /vendor_dlkm/lib/modules. Devices without a vendor_dlkm partition install BOARD_VENDOR_KERNEL_MODULES to /vendor/lib/modules directly. This is problematic as /vendor_dlkm/lib/modules doesn't exist.

Never use /odm_dlkm/lib/modules. Devices without an odm_dlkm partition install BOARD_ODM_KERNEL_MODULES to /odm/lib/modules directly. This is problematic as /odm_dlkm/lib/modules doesn't exist.

Partition mounting and module loading
During first_stage_init, the vendor_dlkm and odm_dlkm partitions are mounted in the /vendor_dlkm and /odm_dlkm directories, respectively. When this happens, symlinks at /vendor/lib/modules and /odm/lib/modules become available.

A vendor process (like an .rc script) can then load the kernel modules based on the order specified in modules.load. The vendor process can also load the modules at a later time, if necessary.

Note: These modules can have load-time dependencies on modules stored in (and loaded from) the vendor boot image.
Related documentation
For documentation regarding the creation of a vendor-boot partition (which contains the vendor RAMDisk), refer to Kernel Module Support.

Here's a simple explanation of **`vendor_dlkm`** and **`odm_dlkm`** partitions in Android, their **purpose**, **structure**, and **how they work**:

✅ TL;DR

`vendor_dlkm.img` and **`odm_dlkm.img`** are **dynamic partitions** introduced in Android 11+ that store **kernel modules** separately from the main **`vendor`** and **`odm`** partitions — so that **you can update modules independently** without touching the whole partition.

🧠 Why were these partitions introduced?

Before Android 11:

- Kernel modules (like **`.ko`** files) were placed **directly in `/vendor/lib/modules`** or **`/odm/lib/modules`**.
- If you wanted to update these modules, you had to update the entire **`vendor`** or **`odm`** partition — which is **risky** and **heavy**.

Starting Android 11:

- To support **modular, independent updates**, all kernel modules are moved to separate dynamic partitions:
    - 📦 **`vendor_dlkm`** → stores vendor modules
    - 📦 **`odm_dlkm`** → stores ODM-specific modules

🏗️ What is inside `vendor_dlkm` and `odm_dlkm`?

These partitions contain:

- `.ko` files (kernel modules)
- `modules.load` (defines the module loading order)
- `modprobe.d` configs (optional)

Mounted Paths:

Partition	Real Path	Mounted At	Symlink Seen From
`vendor_dlkm.img`	`/vendor_dlkm/lib/modules`	`/vendor_dlkm`	`/vendor/lib/modules` → symlink
`odm_dlkm.img`	`/odm_dlkm/lib/modules`	`/odm_dlkm`	`/odm/lib/modules` → symlink

⚙️ Boot-Time Flow

1. `fstab` entry mounts `vendor_dlkm` and `odm_dlkm` at boot in `first_stage_init`.
2. Symlinks like `/vendor/lib/modules` point to real location `/vendor_dlkm/lib/modules`.
3. **Vendor scripts** (e.g., `.rc` files) load modules listed in `modules.load` (using `modprobe` or `insmod`).

🧱 Why symlinks?

> Not all devices have these `*_dlkm` partitions. To keep compatibility:

- Always use `/vendor/lib/modules` or `/odm/lib/modules` in scripts/configs.
- Android creates a symlink to the actual location if `*_dlkm` exists.

📦 How to enable these in build system?

Example (`BoardConfig.mk`)

```make
# Enable vendor_dlkm
BOARD_USES_VENDOR_DLKMIMAGE := true
TARGET_COPY_OUT_VENDOR_DLKM := vendor_dlkm
BOARD_VENDOR_DLKMIMAGE_FILE_SYSTEM_TYPE := ext4
BOARD_SUPER_PARTITION_PARTITION_LIST += vendor_dlkm
```

Add kernel modules

```make
BOARD_VENDOR_KERNEL_MODULES := \
    path/to/foo.ko \
    path/to/bar.ko

# Optional: override load order
BOARD_VENDOR_KERNEL_MODULES_LOAD := path/to/modules.load
```

> At build time:

- Installed to: `$PRODUCT_OUT/vendor_dlkm/lib/modules/`
- Symlink created: `/vendor/lib/modules → /vendor_dlkm/lib/modules`

📂 How do they support OTA updates?

Because `vendor_dlkm` and `odm_dlkm` are **separate dynamic partitions**, they:

- Can be **independently updated via OTA**
- Do **not** require updating `vendor` or `odm` partitions

- Support **Virtual A/B OTA** scheme

In **`device.mk`**:

```
make

AB_OTA_PARTITIONS += vendor_dlkm odm_dlkm
```

💡 Real-world analogy

Think of:

- **`vendor`** as a **sealed box**.
- Kernel modules are **items inside**.
- Without **`vendor_dlkm`**, you'd need to **open the whole box** to update anything inside.
- With **`vendor_dlkm`**, modules are in a **removable side pouch**, so you can **update them independently**.

✅ Summary

Feature	`vendor_dlkm` / `odm_dlkm`
Purpose	Hold kernel modules (as **`.ko`** files) separately
Benefit	OTA updates without touching **`vendor`**/**`odm`**
Mounted At	**`/vendor_dlkm`**, **`/odm_dlkm`**
Seen From	**`/vendor/lib/modules`**, **`/odm/lib/modules`** (symlink)
Android Version	Android 11 and up
Dynamic Partition	Yes (inside **`super.img`**)

Let me know if you'd like:

- 📦 Example **`fstab`** entry
- 🔧 Example **`modules.load`** file
- 📊 Comparison table of all kernel module partitions (**`boot`**, **`vendor_boot`**, **`vendor_dlkm`**, etc.)

18. ⤷
`fstab`
explain simple words

Sure! Let's explain `fstab` in **simple words** with a real-world example:

🔧 What is `fstab`?

`fstab` is a **text file** that tells Android **which partitions to mount**, **where to mount them**, and **how to mount them** during **boot**.

Think of it like a **checklist** or **map** that the Android system follows at boot to attach all the storage areas (like system, vendor, data, etc.).

📦 What does it "mount"?

Mounting means making a storage partition **available** and **accessible** at a folder (mount point) like `/system`, `/vendor`, `/data`.

For example:

- `/vendor` = contains vendor-specific drivers and code
- `/data` = user apps and files
- `/vendor_dlkm` = holds vendor kernel modules

📁 Real Example Entry in `fstab`:

```ini
vendor_dlkm  /vendor_dlkm  ext4  ro,first_stage_mount,logical,avb  wait
```

This line means:

- Mount the partition called `vendor_dlkm`
- Mount it at folder `/vendor_dlkm`
- The partition type is `ext4` (like saying NTFS or FAT in Windows)
- Use these options:
  - `ro` = read-only
  - `first_stage_mount` = mount this **early during boot**
  - `logical` = it's inside **super.img**
  - `avb` = verify its integrity using AVB
  - `wait` = wait for the partition if not ready

🧠 Why is `fstab` important?

Without `fstab`, Android **won't know**:

- Where the system or data lives
- What to mount at boot
- How to mount extra partitions like `vendor_dlkm` or `odm_dlkm`

🏠 Analogy:

Imagine your Android phone as a house.

- `fstab` is like the **floor plan**.
- It tells Android:

  - This is the kitchen (`/system`)
  - This is the garage (`/vendor`)
  - This is the tool shed (`/vendor_dlkm`)
- Without the floor plan, Android doesn’t know **what room is where**, or how to **unlock and access** them.

Let me know if you want a sample `fstab` file explained line-by-line!

19. yes

Perfect! Here's a **sample `fstab` file** (used in Android) and a **line-by-line explanation in simple words**:

📝 Sample `fstab` (from Android AOSP):

```ini
/dev/block/by-name/system   /system        ext4    ro,barrier=1,discard    wait,slotselect,avb
/dev/block/by-name/vendor   /vendor        ext4    ro,barrier=1,discard    wait,slotselect,avb
/dev/block/by-name/userdata /data          ext4    noatime,nosuid,nodev    wait,check,quota
/dev/block/by-name/cache    /cache         ext4    noatime,nosuid,nodev    wait
/dev/block/by-name/metadata /metadata      ext4    noatime,nosuid,nodev    wait,check,formattable
```

🔍 Line-by-Line Breakdown:

1️⃣

```ini
/dev/block/by-name/system   /system   ext4   ro,barrier=1,discard   wait,slotselect,avb
```

- Mount the **system partition** located at **`/dev/block/by-name/system`**
- Mount it at the folder **`/system`**
- Filesystem is **`ext4`**
- Mount options:
  - **`ro`** → Read-Only (can't write)
  - **`barrier=1`** → Ensure write ordering (for safety)
  - **`discard`** → Help manage storage (TRIM support)
- Flags:
  - **`wait`** → Wait until this partition becomes available
  - **`slotselect`** → Choose the correct slot (A/B devices)
  - **`avb`** → Verified with Android Verified Boot

2️⃣

```ini
/dev/block/by-name/vendor   /vendor   ext4   ro,barrier=1,discard   wait,slotselect,avb
```

- Same logic as system
- Mount the **vendor partition** at **`/vendor`**

3️⃣

```ini
/dev/block/by-name/userdata  /data   ext4   noatime,nosuid,nodev   wait,check,quota
```

- Mount **userdata** (apps, downloads, files)
- Mount it at **`/data`**
- Mount options:
  - **`noatime`** → Don't update last-access time (saves writes)

    - **`nosuid`** → Don’t allow SUID programs (security)
    - **`nodev`** → No device files allowed
- Flags:
    - **`wait`** → Wait for partition
    - **`check`** → Run **`fsck`** if needed (check for corruption)
    - **`quota`** → Enable disk quota (limit app usage)

4️⃣

```ini
/dev/block/by-name/cache   /cache   ext4   noatime,nosuid,nodev   wait
```

- Mount **cache partition** (used during update, logs, etc.)
- Mounted at **`/cache`**
- Same options as above
- Simpler flags

5️⃣

```ini
/dev/block/by-name/metadata   /metadata   ext4   noatime,nosuid,nodev   wait,check,formattable
```

- Mount **metadata** partition
- Holds encryption metadata
- **`formattable`** → Can be wiped (for factory reset)

📌 Summary of Common `fstab` Fields

Field	Meaning
`/dev/block/...`	The block device (partition) to mount
`/system`, **`/data`**	Where to mount it in the root filesystem
`ext4`	Filesystem type
`ro`	Read-only
`wait`	Wait until available
`check`	Run fs check (fsck)
`avb`	Use Android Verified Boot
`quota`	Enable disk usage tracking
`formattable`	Can be erased if needed
`slotselect`	Used in A/B system updates

Let me know if you'd like a breakdown of **`fstab`** used in **`vendor_boot`**, or how **`fstab`** differs between **first-stage init vs second-stage**!

20. Android shared system image

bookmark_border
This page presents several mechanisms that Android device OEMs can use to have their own shared system image (SSI) across product lines. It also proposes a procedure for basing an OEM-owned SSI on an AOSP-built generic system image (GSI).

Background
With Project Treble, monolithic Android was split into two parts: the hardware-specific part (the vendor implementation) and the generic OS part (the Android OS framework). The software for each is installed in a separate partition: the vendor partition for the hardware-specific software, and the system partition for the generic OS software. A versioned interface, called the vendor interface (VINTF), is defined and enforced across the two partitions. By using this partitioning system, you can modify the system partition without modifying the vendor partition, and vice versa.

Motivation
The framework code released in AOSP has been compliant with the Treble architecture and has maintained backward compatibility with older vendor implementations. For example, a generic system image built from Android 10 AOSP sources can run on any Treble-compliant device that's running on Android 8 or higher. The version of Android that's shipped on consumer devices is modified by SoC vendors and OEMs. (See Life of an Android Release.) These changes and extensions that were made to the framework weren't written for maintaining backward compatibility, which translated to increased complexity and higher cost in an OS upgrade. Device-specific changes and modifications add to the cost and complexity of upgrading an Android OS version.

Before Android 11 there was no clear architecture that enabled partners to build modular extensions to the Android OS framework. This document describes the steps that SoC vendors and OEMs can take to create an SSI. This means one image, built from the Android OS framework sources for reuse across multiple devices, for maintaining backward compatibility with vendor implementations, and for providing a significant reduction in the complexity and cost of Android OS upgrades. For the specific steps you need to create an SSI, see the Suggested steps for GSI-based SSI section, and note that you don't have to use all four steps. Which steps you choose (only Step 1, for example) depends on your implementation.

SSI overview
With SSI, product-specific software components and OEM extensions are placed in a new /product partition. The components in the /product partition use a well-defined, stable interface to interact with components in the /system partition. OEMs can either choose to build one SSI, or to have a small number of SSIs for use across multiple device SKUs. When a new version of the Android OS is released, OEMs invest only once in updating their SSIs to the latest Android release. They can reuse the SSIs to update multiple devices without updating the /product partition.

Note that OEMs and SoC vendors build SSIs that include all the custom features and modifications that an OEM needs. The mechanisms and best practices provided on this page are intended for OEMs to use to reach these key goals:

Reuse the SSI across multiple device SKUs.
Update the Android system with the modular extensions to make OS upgrades easier.
The core idea of separating product-specific components into the product partition is similar to the Treble idea of separating SoC-specific components into the vendor partition. A product interface (similar to VINTF) allows communication between SSI and the product partition. Note that with respect to SSI, the term "components" describes all the resources, binaries, texts, libraries, and so on that are installed to images, which essentially become partitions.

Here's a **simple explanation** of the **Android Shared System Image (SSI)** concept and why it matters:

✅ What is SSI (Shared System Image)?

An **SSI** is a **shared version of the Android system partition** (i.e., the core Android OS) that OEMs (like Samsung, Xiaomi, etc.) can **reuse across many devices**.

Think of it as **"One Android OS build → used on many devices"** by the same manufacturer.

🔧 Why is this needed?

Before SSI:

- Each device model had its own slightly different system image.
- Updating Android meant updating dozens of different images → **high cost, hard work**.

With SSI:

- One common system image (SSI) works across many phones.
- You only need to update this SSI once for many devices → **saves effort**.

🧱 How does SSI work?

Android splits system components into partitions:

- `/system` → Contains the **generic Android OS framework** (the SSI)
- `/vendor` → Contains **hardware-specific** things (from SoC vendors like Qualcomm)
- `/product` → Contains **device-specific customizations** (from OEMs like Samsung)

With SSI:

- You put your **custom features into** `/product`
- You build **one standard** `/system` **image (SSI)** for all devices
- These parts **communicate using stable interfaces**, like **VINTF** and **Product Interface**

🧠 Think of it like this:

Component	Role	Owned by
`/system`	Android OS Core (SSI)	Google/OEM
`/vendor`	Hardware drivers, HALs	SoC vendor
`/product`	Custom apps, features, UI skins	OEM

🛠️ Key Benefits

Benefit	Description
🔁 **Reusability**	One SSI for many devices
🚀 **Faster OS updates**	Update `/system` once, reuse it
💰 **Lower cost**	Less testing, less duplication
✅ **Modularity**	Clean separation between system, vendor, and product
📱 **Better maintenance**	Easier to fix bugs, support longer lifecycles

---

📦 SSI is similar to:

- **Treble** split `/system` and `/vendor`
- **SSI** adds one more level: `/product`

Now, each layer (SoC, Android OS, OEM customizations) has its own partition, **making the system modular and easier to manage**.

📌 Example

Imagine Xiaomi wants to release 10 phones:

Without SSI:

- Build 10 separate system images (slow and painful)

With SSI:

- Build 1 SSI (`/system`)
- Add device-specific features in `/product` for each phone

Now, when Android 15 comes out:

- Just update **one SSI**
- All 10 phones get the update faster

Would you like a **diagram** to visualize this modular split of system/vendor/product/SSI?

21. it came after treble?

Yes, the **Shared System Image (SSI)** concept came **after Project Treble** — it is essentially a **next step building on Treble's foundation**.

🔁 Timeline of Changes

Feature/Architecture	Android Version	Purpose
Project Treble	Android 8.0	Split Android into `/system` and `/vendor` to separate hardware-specific and OS framework components
VNDK / VINTF	Android 8.0+	Ensure stable interface (API/ABI) between `/system` and `/vendor`
GSI (Generic System Image)	Android 9+	A pure AOSP system image to test Treble compatibility
SSI (Shared System Image)	Android 11+	Enables OEMs to share one `/system` image across devices, with customizations moved to `/product`

🧠 In Simple Words:

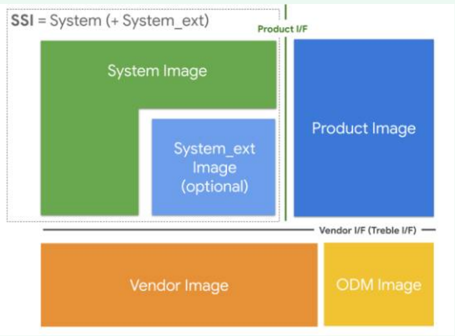
- **Treble** split Android into two:
  → `/system` (OS) and `/vendor` (hardware)
- **SSI** said:
  → "Let's also move OEM-specific customizations out of `/system` and into `/product`, so we can reuse `/system` too."

This makes upgrades even easier:

- Now, **hardware** lives in `/vendor`
- **OEM customizations** in `/product`
- And the **generic Android OS** in `/system` can be **shared and updated easily**

Let me know if you'd like a **diagram** showing the evolution from pre-Treble to Treble to SSI.

22.



Partitions around SSI
Figure 1 shows partitions around SSI, and the versioned interfaces across the partitions and policies on the interfaces. This section explains each of the partitions and interfaces in detail.

Images and partitions
The information in this section distinguishes between the terms image and partition.

An image is a conceptual piece of software that can be updated independently.
A partition is a physical storage location that can be updated independently.
The sections in Figure 1 are defined as follows:

SSI: The SSI is the image that's common to an OEM, and can exist across multiple devices. It doesn't have any hardware-specific or product-specific components. Everything in a given SSI is, by definition, shared among all devices using that SSI. The SSI is composed of either a single /system image, or a /system and the /system_ext partitions, as seen in Figure 1.

The /system partition contains AOSP-based components, while /system_ext, when implemented, contains OEM and SoC vendor extensions and components that are tightly coupled with AOSP components. For example, an OEM Java framework library that provides custom APIs for the OEM's own apps fits better in the /system_ext than in the /system partition. Content for both the /system and /system_ext partitions is built from OEM-modified Android sources.

The /system_ext partition is optional, but it's beneficial to use it for any custom features and extensions that are tightly coupled with AOSP- based components. This distinction helps you identify changes you need to make, to move such components from the /system_ext partition to the /product partition over a period of time.

Product: A collection of product- or device-specific components that represent OEM customizations and extensions to the Android OS. Put SoC-specific components in the /vendor partition. SoC vendors can also use the /product partition for appropriate components, such as SoC-independent ones. For example, if an SoC vendor provides an SoC-independent component to their OEM customers (that's optional to ship with the product), the SoC vendor can place that component in the product image. The location of a component isn't determined by its ownership, it's dictated by its purpose.

Vendor: A collection of SoC-specific components.

ODM: A collection of board-specific components that aren't provided by the SoC. Typically the SoC vendor owns the vendor image, while the device maker owns the ODM image. When there is no separate /odm partition, both the SoC vendor and ODM images are merged together in the /vendor partition.

Interfaces between images
Two main interfaces for vendor and product images exist around SSI:

Vendor Interface (VINTF): VINTF is the interface to the components that reside in the vendor and the ODM images. Components in the product and system images can only interact with the vendor and ODM images through this interface. For example, a vendor image can't depend on a private part of the system image, and vice versa. This is originally defined in Project Treble, which split the images into system and vendor partitions. The interface is described using the following mechanisms:

HIDL (Passthrough HAL is only available for system and system_ext modules)
Stable AIDL
Configurations
System properties API
Config file schema API
VNDK

Android SDK APIs
Java SDK library
Product interfaces: The product interface is the interface between SSI and the product image. Defining a stable interface decouples the product components from the system components in an SSI. The product Interface requires the same stable interfaces as VINTF. However, only the VNDK and Android SDK APIs are enforced for devices launching with Android 11 (and higher).

Enable SSI in Android 11
This section explains how to use the new features in place to support SSI in Android 11.

The /system_ext partition
The /system_ext partition was introduced in Android 11 as an optional partition. (It's the place for non-AOSP components that have tight coupling with the AOSP-defined components in the /system partition.) The /system_ext partition is assumed to be the OEM-specific extension to the /system partition, without an interface defined across the two partitions. Components in the /system_ext partition can make private API calls into the /system partition, and components in the /systempartition can make private API calls into the /system_ext partition.

Because the two partitions are tightly coupled, both partitions are upgraded together when a new Android version is released. A /system_ext partition created for the previous release of Android doesn't need to be compatible with the /system partition in the next Android release.

To install a module to the /system_ext partition, add system_ext_specific: trueto the Android.bp file. For devices that don't have a /system_ext partition, install such modules to the ./system_ext subdirectory in the /system partition.

History
Here is some history about the /system_ext partition. The design goal was to place all OEM-specific components, regardless of whether they're common, in the /product partition. However, moving them all at once wasn't feasible, especially when some components had a tight coupling with the /system partition. To move a tightly coupled component to the /product partition, the product interface must be extended. This often required the component itself to be extensively refactored, which consumes a lot of time and effort. The /system_ext partition started as a place to temporarily host those components that aren't ready to be moved to the /product partition. The goal of the SSI was to eventually eliminate the /system_ext partition.

However, the /system_ext partition is useful for keeping the /system partition as close to AOSP as possible. With SSI, most of the upgrade effort is spent on the components in the /system and the /system_ext partitions. When the system image is built from sources that are as similar as possible to those in AOSP, you can focus the upgrade effort on the system_ext image.

Unbundle components from /system and /system_ext partitions into the /product partition
Android 9 introduced a /product partition that's coupled with the /system partition. The modules in the /product partition use the system resources without any restriction, and vice versa. To make SSI possible in Android 10, the product components are split into the /system_ext and /product partitions. The /system_ext partition doesn't have to adhere to the restrictions on using system components that the /product partition did in Android 9. Starting in Android 10, the /product partition must be unbundled from the /system partition and must use stable interfaces from the /system and /system_ext partitions.

The /system_ext partition's primary purpose is to extend system features, rather than to install bundled product modules, as described in the /system_ext partition section. To do this, unbundle the product-specific modules and move them into the /product partition. Unbundling the product-specific modules makes /system_ext common to the devices. (For more detail, see Making the /system_ext partition common.)

To unbundle the /product partition from the system components, the /product partition must have the same enforcement policy as the /vendor partition that was already unbundled with Project Treble.

Starting in Android 11, native and Java interfaces for the /product partition are enforced as described below. For more information, see Enforcing Product Partition Interfaces.

Native interfaces: The native modules in the /product partition must be unbundled from the other partitions. The only allowed dependencies from the product modules are some VNDK libraries (including LLNDK) from the /system partition. JNI libraries that the product apps depend on must be NDK libraries.
Java interfaces: The Java (app) modules in the /product partition can't use hidden APIs, because they're unstable. These modules must only use public APIs and system APIs from the /system partition, and Java SDK libraries in the /system or /system_ext partition. You can define Java SDK libraries for custom APIs.

Great! You've uploaded the diagram showing **partitions and interfaces around SSI (Shared System Image)**. Let me now explain everything shown in the image in **very simple words**, step by step:

🧩 What the Diagram Shows

🟩 SSI (Shared System Image)

- **Contains:**
  - `/system` → Android OS framework from AOSP.
  - `/system_ext` (optional) → OEM extensions that are *tightly coupled* with `/system` (e.g., custom Java frameworks or HAL passthroughs).
- **Purpose:** Reuse the same system image across multiple devices (cheaper and easier upgrades).
- **Internal Relationship:** `/system` and `/system_ext` are closely connected and upgraded together.

🔵 `/product` → Product Image

- **What's inside:** Device-specific or product-specific apps, binaries, resources (e.g., OEM launchers, camera apps).
- **Interacts with:** SSI (via **Product Interface**)
- **Must use:** Only **stable public APIs** from `/system` and `/system_ext` (no private or hidden APIs allowed)

🟧 `/vendor` → Vendor Image

- **What's inside:** SoC-specific drivers, HALs, firmware blobs (from Qualcomm, MediaTek, etc.)
- **Interacts with:** SSI via **VINTF (Vendor Interface)**
- **Follows rules of:** Project Treble

🟨 `/odm` → ODM Image

- **What's inside:** Board/device-specific HALs and drivers (added by the OEM/ODM)
- **Can be merged with:** `/vendor` if `/odm` is missing
- **Follows same rules:** As `/vendor` (uses VINTF)

🔗 Interfaces

1. Vendor Interface (VINTF)

Defined in **Project Treble**

- Controls interaction between `/system[_ext]` and `/vendor//odm`
- Examples:
  - HAL interfaces (HIDL, AIDL)
  - System properties, config files
  - VNDK libraries (for native code)

2. Product Interface

Newer (introduced with **SSI**)

- Controls interaction between `/system[_ext]` and `/product`
- Enforces:
  - Use of stable APIs only (like vendor interface)
  - No hidden/private access
  - Only public Java APIs or defined SDKs

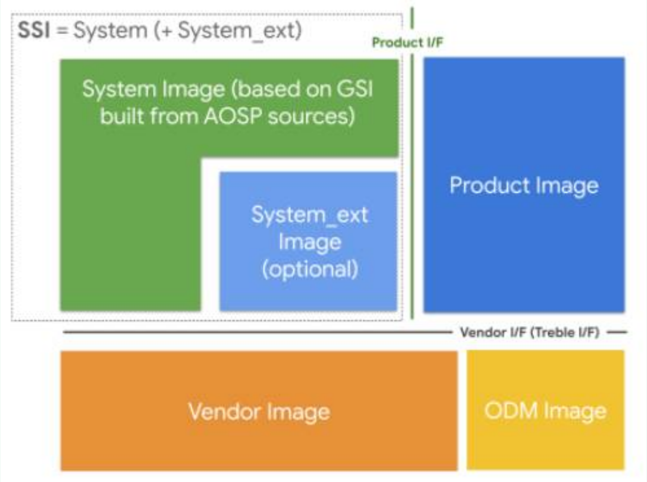
✅ Summary: Why It Matters

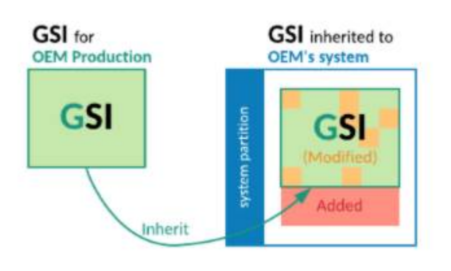
Goal	How SSI + Partitions Help
Faster upgrades	Reuse `/system` & `/system_ext` across many devices
Better modularity	Separate OEM code (`/product`) and SoC code (`/vendor`)
Fewer bugs and breakages	Force use of stable APIs across partitions

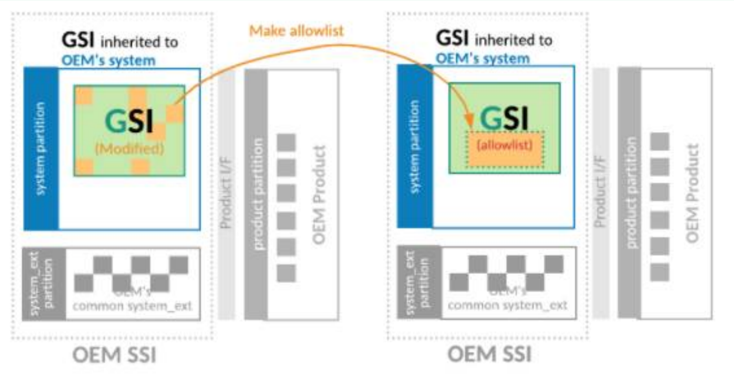
Goal	How SSI + Partitions Help
Smaller OTA updates	Can update partitions independently

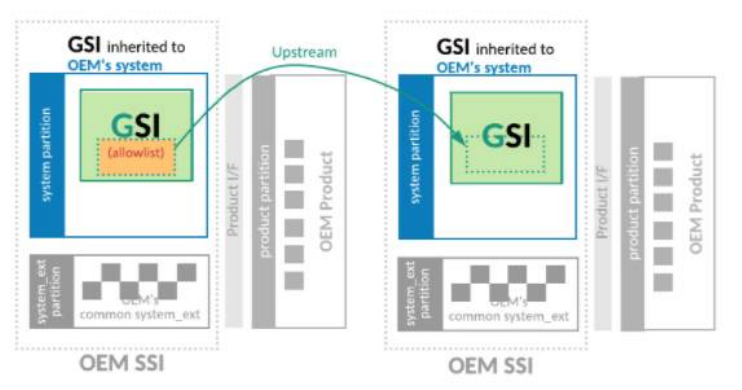
Would you like a **custom-drawn simplified version of this image** or a **step-by-step boot-time flow** using SSI structure?

23.


SSI = System (+ System_ext)
Product I/F
System Image (based on GSI built from AOSP sources)
System_ext Image (optional)
Product Image
Vendor I/F (Treble I/F)
Vendor Image
ODM Image







A generic system image (GSI) is the system image that's built directly from AOSP. It's used for the Treble compliance tests (for example, CTS-on-GSI) and as a reference platform that app developers can use to test the compatibility of their apps when they don't have a real device running the required version of Android.

OEMs can also use GSI to make their SSI. As explained in Images and partitions, SSI consists of the system image for the AOSP-defined components and the system_ext image for the OEM-defined components. When GSI is used as the system image, the OEM can focus on the system_ext image for the upgrade.

This section provides a guide to OEMs who want to modularize their customizations into the /system_ext and /product partitions while using an AOSP or near-AOSP system image. If OEMs build the system image from AOSP sources, then they can substitute the system image that they build with the GSI provided by AOSP. However, OEMs don't need to reach the final step (using GSI as it is) all at once.

Step 1. Inherit generic_system.mk for OEM's system image (OEM GSI)
By inheriting generic_system.mk (which was named mainline_system.mk in Android 11, and renamed to generic_system.mk in AOSP), the system image (OEM GSI) includes all the files that the AOSP GSI has. These files can be modified by OEMs, so that the OEM GSI can contain the OEM proprietary files in addition to the AOSP GSI files. However, OEMs aren't allowed to modify the generic_system.mk file itself.

Step 2. Make the OEM GSI have the same list of files with the AOSP GSI
The OEM GSI can't have additional files at this stage. The OEM's proprietary files must be moved out to the system_ext or product partitions.

Step 3. Define an allowlist to limit the modified files in the OEM GSI
To check the modified files, OEMs can use the compare_images tool, and compare the AOSP GSI with the OEM GSI. Obtain the AOSP GSI from the AOSP lunch target generic_system_*.

By running the compare_images tool periodically with the allowlist parameter, you can monitor the differences outside the allowed list. This prevents needing additional modifications to the OEM GSI.

Step 4. Make the OEM GSI have the same binaries as the AOSP GSI
Cleaning up the allowlist allows OEMs to use the AOSP GSI as the system image for their own products. To clean up the allowlist, OEMs can either abandon their changes in the OEM GSI, or upstream their changes to AOSP so that the AOSP GSI includes their changes.

Define SSI for OEMs
Protect the /system partition at build time
To avoid any product-specific changes in the /system partition and define the OEM GSI, OEMs can use a makefile macro called require-artifacts-in-path to prevent any declaration of system modules after the macro is called. See the Create makefile and enable artifact path check example.

OEMs can define a list to allow product-specific modules to be installed in the /system partition temporarily. However, the list must be empty to make the OEM GSI common to all of the OEM's products. This process is for defining the OEM GSI and can be independent from the steps for the AOSP GSI.

Enforce product interfaces
To guarantee that the /product partition is unbundled, OEMs can ensure their devices enforce the product interfaces by setting PRODUCT_PRODUCT_VNDK_VERSION:= current for native modules, and PRODUCT_ENFORCE_PRODUCT_PARTITION_INTERFACE:= true for Java modules. These variables are automatically set if the PRODUCT_SHIPPING_API_LEVEL of the device is greater than or equal to 30. For detailed information, see Enforcing Product Partition Interfaces.

Make the /system_ext partition common
The /system_ext partition might differ between devices, because it can have device-specific, system-bundled modules. Because the SSI consists of /system and /system_ext partitions, the differences in the /system_ext partition hinder OEMs from defining an SSI. OEMs can have their own SSI and can share that SSI among multiple devices by removing any differences and making the /system_ext partition common.

This section gives recommendations for making the /system_ext partition common.

Expose hidden APIs in the system partition
Many product-specific apps can't be installed in the product partition because they use hidden APIs, which are prohibited in the product partition. To move device-specific apps to the product partition, remove the use of hidden APIs.

The preferred way to remove hidden APIs from the apps is to find the alternative public or system APIs to replace them. If there are no APIs to replace the hidden APIs, OEMs can contribute to AOSP to define the new system APIs for their devices.

Alternatively, OEMs can define custom APIs by creating their own Java SDK library in the /system_ext partition. It can use hidden APIs in the system partition, and can provide the APIs to the apps in the product or vendor partition. OEMs must freeze the product-facing APIs for backward compatibility.

Include the superset of all APKs and skip some packages installs for each device
Certain packages that are bundled with the system aren't common across devices. Unbundling these APK modules to move them to the product or the vendor partition can be difficult. As an interim solution, OEMs can make the SSI include all the modules, then filter unwanted ones out by using a SKU property (ro.boot.hardware.sku). To use the filter, OEMs overlay the framework resources config_disableApkUnlessMatchedSku_skus_list and config_disableApksUnlessMatchedSku_apk_list.

For more precise settings, declare a broadcast receiver that disables unnecessary packages. The broadcast receiver calls setApplicationEnabledSetting to disable the package when it receives the ACTION_BOOT_COMPLETED message.

Define RRO instead of using static resource overlay
A static resource overlay manipulates the overlaid packages. However, it can impede defining an SSI, so ensure that properties for RRO are turned on and set properly. By setting the properties as follows, OEMs can have all auto-generated overlays as RROs.

PRODUCT_ENFORCE_RRO_TARGETS := *
PRODUCT_ENFORCE_RRO_EXCLUDED_OVERLAYS := # leave it empty
If a detailed configuration is required, define an RRO manually instead of relying on an auto-generated one. For detailed information, see Runtime Resource Overlays (RROs). OEMs also can define conditional RROs that depend on the system properties by using the android:requiredSystemPropertyName and android:requiredSystemPropertyValue attributes.

Frequently asked questions (FAQ)
Can I define multiple SSIs?
It depends on the commonality and characteristics of devices (or device group). OEMs can try to make the system_ext partition common, as described in Making the system_ext partition common. If a device group has many differences, then it's better to define multiple SSIs.

Can I modify generic_system.mk (mainline_system.mk) for an OEM GSI?
No. But OEMs can define a new makefile for an OEM GSI that inherits the generic_system.mk file and use the new makefile instead. For an example, see Enforcing Product Partition Interfaces.

Can I remove modules from generic_system.mk that conflict with my implementation?
No. GSI has a minimum set of bootable and testable modules. If you think a module isn't essential, please file a bug to update the generic_system.mk file in AOSP.

Yes, Shared System Image (SSI) **came after Treble**.

Let me explain this in **very simple words**, including the meaning of the diagrams you uploaded:

🧠 First, Understand Treble (Android 8+):

- Treble **split Android into two parts**:
  1. **System (OS)** → Android framework code (`/system` partition)
  2. **Vendor** → Hardware-specific drivers and code (`/vendor` partition)

🧩 These are connected by a **stable interface** called **VINTF (Treble I/F)**.

🆕 What is SSI (Shared System Image)?

SSI is a concept **introduced after Treble**, around **Android 10/11**, to:

✅ **Share the same system image** across many devices of the same OEM (e.g., all Samsung phones can use one base system image)

✅ **Reduce upgrade effort** (you update the system image once, reuse it on all models)

📦 SSI Includes:

- `/system` → Pure Android OS (GSI or near-GSI)
- `/system_ext` → Optional OEM extensions (e.g., special APIs)

🧱 These two together form **SSI = /system + /system_ext**

📦 Other Partitions Around SSI:

1. `/product` → Device-specific stuff (OEM apps, features)
2. `/vendor` → SoC or chipset-specific drivers (GPU, modem, etc.)
3. `/odm` → ODM-specific stuff (board-level details)

Each part talks via:

- **Product I/F** between `/system` and `/product`

- **Vendor I/F (Treble)** between `/system` and `/vendor`

🖼️ Diagrams Explained:

✅ 1st Diagram (SSI = System + System_ext):

Shows SSI and its neighbors: `/product`, `/vendor`, `/odm`.

✅ **Purpose**: Modularize system so OEMs can update it without touching vendor code.

✅ 2nd Diagram (GSI → OEM system):

- OEM starts with **GSI from AOSP** (clean Android)
- OEM adds its code on top to make **Modified GSI**
- Goal: **gradually reduce changes**, become closer to GSI

✅ 3rd Diagram (Make allowlist):

- If OEM has changed some GSI parts, they keep track in an **allowlist** (list of allowed customizations)
- This helps ensure fewer hidden/accidental changes

✅ 4th Diagram (Upstream or eliminate allowlist):

- Eventually, OEM can:
  - Upstream changes to AOSP
  - Or remove them
- Then OEM can use **pure GSI** as their system image

🎯 **Goal**: Make OEM GSI = AOSP GSI

✅ Steps to Reach Pure SSI (Summary):

1. Inherit from AOSP's `generic_system.mk`
2. Move OEM-specific code from `/system` → `/system_ext` or `/product`
3. Use a **file allowlist** to track differences
4. Eventually use **AOSP GSI directly** as system image (SSI)

💡 Why is this Useful?

- Makes updates **faster and cheaper**
- OEMs can maintain **1 system image for many devices**
- Encourages **cleaner architecture** and long-term maintainability

---

Would you like me to make a diagram with simplified colors to explain this process visually?

24. GSI vs SSI

Here's a **clear and simple comparison between GSI and SSI** — what they are, how they are used, and how they relate:

✅ GSI (Generic System Image)

Aspect	Description
What is it?	A pure Android system image built directly from **AOSP** sources.
Purpose	- Used for **testing Treble compliance** (CTS-on-GSI). - Used by developers to test apps on **reference Android**. - Acts as the **base image** for creating SSI.
Who uses it?	- Google (for testing) - App developers - OEMs (as a starting point to make SSI)
Customization	Not customized. **No OEM or hardware-specific code**.
Includes	Only `/system` partition, built strictly from AOSP.
Update	Controlled by AOSP. OEMs don't change it (unless for internal use).

✅ SSI (Shared System Image)

Aspect	Description
What is it?	A system image **shared by an OEM** across multiple devices. Built from AOSP (like GSI) but **may include minimal OEM additions**.
Purpose	- **Used by OEMs** in production across multiple devices. - Reduces upgrade cost by sharing one system image.
Who uses it?	OEMs (Samsung, Xiaomi, etc.) to ship Android on many devices.
Customization	Slightly modified from GSI (at first), then **ideally becomes close or equal to GSI**. OEM-specific code moved to `/system_ext` or `/product`.
Includes	`/system` (from GSI) + optional `/system_ext` (OEM extensions) = SSI
Update	OEM-controlled, but ideally **based on GSI** to ease upgrades.

🔁 Relationship between GSI and SSI

1. **GSI** is the **starting point** (from AOSP).
2. OEM makes small changes or additions to GSI.
3. OEM puts custom parts into `/system_ext` or `/product`.
4. The final result = **SSI** → shared across devices.
5. Over time, SSI can be made to **match GSI** completely.

📊 Summary Table

Feature	GSI	SSI
Built from AOSP	✅ Yes	✅ Yes (inherits GSI)
OEM Custom Code	❌ None	✅ In `/system_ext` or `/product`
Used for	Testing, compliance	Actual device deployment
Shared Across Devices	❌ No	✅ Yes
Target Audience	Google, devs, OEM testing	OEMs for product lines
Can be updated independently	✅ Yes	✅ Yes

📘 In Simple Words:

- **GSI** is like a **pure Android blueprint**.

- **SSI** is like an **Android OS version an OEM builds once and uses on many phones**.
- **GSI helps build SSI**, and **SSI helps simplify Android updates**.

Want a custom diagram showing how GSI transforms into SSI across OEM devices?

25. ↪
Over time, SSI can be made to **match GSI** completely.
Explain this statement

Great question!

🔹 Statement:

> "**Over time, SSI can be made to match GSI completely.**"

🔸 Simple Explanation:

This means that an **OEM's Shared System Image (SSI)** — which may **initially have some custom changes** — can eventually be cleaned up so that it **becomes identical to Google's Generic System Image (GSI)**.

🔸 Why is this useful?

If the **SSI becomes identical to the GSI**:

- The OEM **doesn't need to maintain a custom system image**.
- The OEM only needs to focus on `/system_ext` and `/product` partitions.
- It **greatly simplifies Android upgrades** — just drop in the latest GSI as-is.
- Ensures **maximum compatibility** and fewer bugs across devices.

🔸 How does this happen "over time"?

At first:

- OEM adds custom features directly into `/system`.

Then gradually:

1. OEM **moves those features out** of `/system` into `/system_ext` or `/product`.
2. They **stop modifying** the core system files from AOSP.
3. Eventually, their SSI becomes **equal to GSI** — no differences.

🔸 Analogy:

Imagine GSI is a clean base Android pizza.

At first, OEM adds some toppings directly on the base (system image).

But to make the pizza upgradeable easily, the OEM:

- Moves toppings to a side container (`/product` or `/system_ext`)
- Leaves the base untouched (same as GSI)

Eventually, they just reuse the original base (GSI) and **add toppings separately**.

Would you like to see a diagram or step-by-step SSI cleanup flow?

26. In what sense SSI helping OEM?

Great question!

✅ How SSI helps OEMs (Original Equipment Manufacturers):

A **Shared System Image (SSI)** is a big help for OEMs in several ways — mostly by reducing **cost, complexity, and time** in development and upgrades across devices.

🔹 1. Reuse Across Devices (SKUs)

- OEMs often launch **many devices** (SKUs) with similar software.
- SSI lets them **share the same `/system` (+ optional `/system_ext`) image** across devices.
- That means **no need to build different system images for every device**.

🟢 **Benefit**: Less duplication → faster development and testing.

🔹 2. Simplified Android OS Upgrades

- When the `/system` is shared and standardized, upgrading to a new Android version (e.g., Android 14 → 15) means:
  - **Update once**, reuse across devices.
  - No need to modify `/system` separately for each product.

🟢 **Benefit**: Saves time, effort, and resources during Android version upgrades.

🔹 3. Modularization of Features

- OEM-specific code moves to `/system_ext` or `/product`, separate from the AOSP `/system`.
- This **decouples device-specific and OEM-specific logic** from the AOSP system.

🟢 **Benefit**: Easier to maintain and debug device-specific features.

🔹 4. Improved Compliance and Stability

- By aligning more closely with AOSP (or even GSI), OEMs:
  - Reduce compatibility issues.
  - Pass CTS-on-GSI tests more easily.
  - Build more stable platforms.

🟢 **Benefit**: Fewer bugs, better app compatibility, faster certification.

🔹 5. Lower Maintenance Cost

- Single SSI image = fewer test matrices.
- Less engineering time spent maintaining multiple branches or fixing conflicts in system code.

🟢 **Benefit**: Significant cost savings, especially for large OEMs.

🔹 6. Faster Feature Rollouts

- Since the base (SSI) is shared and frozen, teams can **develop and test product features in parallel** using the same system image.

🟢 **Benefit**: Speeds up time-to-market.

Summary Table:

Advantage	Why it helps OEMs
Reuse across devices	One image, many devices
Easier Android upgrades	No need to rework for each device
Cleaner modular design	Clear separation of concerns
Better GSI compatibility	Easier testing and certification
Lower engineering cost	Less duplication of work
Faster development	Shared base lets teams work in parallel

Would you like a visual/diagram to explain this as well?

27. Ramdisk partitions

bookmark_border
Note: This page applies to Android 11 and lower. For Android 12 and greater, see Generic boot partition.
In Android 10 and 11, the first stage ramdisk contains the first stage init binary (which performs early mounting as specified by fstab entries) and vendor fstab files. (As in Android 9 and lower, system.img contains the contents of $TARGET_ROOT_OUT.)

For devices with a boot-ramdisk (non-A/B), first stage init is a static executable located at /init. These devices mount system.img as /system, then perform a switch root operation to move the mount at /system to /. The contents of the ramdisk are freed after mounting has completed.

For devices that use recovery as a ramdisk, first stage init is located at /init within the recovery ramdisk. These devices first switch root to /first_stage_ramdisk to remove the recovery components from the environment, then proceed the same as devices with a boot-ramdisk (that is, mount system.img as /system, switch root to move that mount to /, and free ramdisk contents after mounting). If androidboot.force_normal_boot=1 is present in the kernel command line (or bootconfig from Android 12 onward), devices boot normally (into Android) instead of booting into recovery mode.

After first stage init finishes, it executes /system/bin/init with the selinux_setup argument to compile and load SELinux onto the system. Finally, init executes /system/bin/init again with the second_stage argument. At this point, the main phase of init runs and continues the boot process using the init.rc scripts.

Note: Android 10 replaces ramdisk contents with a first stage init executable, which is incompatible with how previous ramdisks booted the system.
The following sections detail differences in partition layouts for non-A/B devices before and after Android 10.

Partition layouts (non-A/B devices)
The following sections detail differences in partition layouts for non-A/B devices before and after Android 10.

boot.img
Ramdisk
(Android 8.x and lower) System as root
(Android 9) Ramdisk
(Android 10 and 11)
Contains a kernel and a ramdisk.img.

ramdisk.img
 -/
  - init.rc
  - init
  - etc -> /system/etc
  - system/ (mount point)
  - vendor/ (mount point)
  - odm/ (mount point)
  ...

Contains a normal boot kernel only.
Contains a kernel and ramdisk.img.

ramdisk.img
 -/
  - init
  - vendor fstab files
  - system/ (mount point)
  - vendor/ (mount point)
  - odm/ (mount point)
  ...

recovery.img
Contains a recovery kernel and a recovery ramdisk.img

system.img
Ramdisk

(Android 8.x and lower)   System as root
(Android 9)   Ramdisk
(Android 10 and 11)
Contains a system.img.

```
system.img
  -/
    - bin/
    - etc
    - vendor -> /vendor
    - ...
```

Contains the merged contents of $TARGET_SYSTEM_OUT and $TARGET_ROOT_OUT.

```
system.img
  -/
    - init.rc
    - init
    - etc -> /system/etc
    - system/
      - bin/
      - etc/
      - vendor -> /vendor
      - ...
    - vendor/ (mount point)
    - odm/ (mount point)
    ...
```

Contains the merged contents of $TARGET_SYSTEM_OUT and $TARGET_ROOT_OUT.

```
system.img
  -/
    - init.rc
    - init -> /system/bin/init
    - etc -> /system/etc
    - system/
      - bin/
      - etc/
      - vendor -> /vendor
      - ...
    - vendor/ (mount point)
    - odm/ (mount point)
    ...
```

Partition layouts (A/B devices)
The following sections detail differences in partition layouts for A/B devices before and after Android 10.

boot.img
System as root
(Android 9)   Ramdisk
(Android 10 and 11)
Contains normal boot kernel and recovery-ramdisk (BOARD_USES_RECOVERY_AS_BOOT := true).

Recovery-ramdisk is used only to boot into recovery. Contains normal boot kernel and recovery-ramdisk (BOARD_USES_RECOVERY_AS_BOOT := true).

Recovery-ramdisk is used to boot into both recovery and Android.

```
ramdisk.img
  -/
    - init -> /system/bin/init
    - first_stage_ramdisk
       - vendor fstab files
    - etc -> /system/etc
    - system/ (mount point)
    - vendor/ (mount point)
    - odm/ (mount point)
    ...

system.img
Contains the merged contents of $TARGET_SYSTEM_OUT and $TARGET_ROOT_OUT.


system.img
  -/
    - init.rc
    - init -> /system/bin/init
    - etc -> /system/etc
    - system/
      - bin/
      - etc/
      - vendor -> /vendor
      - ...
    - vendor/ (mount point)
    - odm/ (mount point)
    ...
```

Here's a **simple explanation** of the **Ramdisk partition** changes in Android (especially around Android 10 and 11), and how **boot.img, ramdisk, and system.img** are laid out:

🔹 What is Ramdisk?

- It's a temporary filesystem loaded into **RAM** during early boot.
- It contains the **first stage init process** (`/init`), and **fstab** files to mount the actual partitions like `/system`, `/vendor`, `/odm`.

🔹 Key Timeline (Android 8 → 11)

Android Version	Ramdisk Content	Key Change
Android 8.x↓	`init.rc`, `init`, etc. in ramdisk	`system` not mounted as `/` yet
Android 9	**System-as-root** introduced	`system.img` is mounted as `/`
Android 10–11	**First-stage init** in ramdisk	Ramdisk only used early in boot

---

🔹 Two Device Types

1. Non-A/B Devices

> These have separate boot and recovery partitions.

- **boot.img** contains:
  - **Kernel**
  - **Ramdisk.img**
    - `/init`
    - `fstab` files
    - mount points: `/system`, `/vendor`, `/odm`
- The **system.img**:
  - Contains full OS (merged `$TARGET_SYSTEM_OUT + $TARGET_ROOT_OUT`)
  - Mounted as `/` (root) after early boot.
- Ramdisk is **discarded after boot**.

---

2. A/B Devices (Seamless Updates)

> No separate recovery.img; boot.img is used for both normal and recovery boot.

- **boot.img**:
  - Includes a **"recovery-ramdisk"** used **even for normal Android boot**.
  - It sets up `/first_stage_ramdisk`, then system.img is mounted.
- Inside **ramdisk**:

```swift
/
  init -> /system/bin/init
  first_stage_ramdisk/
    vendor fstab files
  system/
  vendor/
  odm/
```

🔹 How Boot Works (Simplified)

Step-by-step boot:

1. **Kernel loads boot.img**, extracts ramdisk
2. **First-stage init** in ramdisk runs:
   - Mounts `/vendor`, `/odm`, `/system`
   - Switches root (`pivot_root`) to `/system`
3. Loads **SELinux**, then starts second stage init
4. Final system boots from system.img (which is now `/`)

🔹 Why This Matters

- Splitting logic into **ramdisk vs system** helps:
  - Boot faster
  - Support **System-as-root**
  - Support **A/B partitions** (seamless OTA updates)
  - Modular boot: first stage is **minimal** (only mounts), rest happens later

---

🔹 Summary Table

Component	Pre-Android 10	Android 10/11
Ramdisk	Big, has init.rc, etc.	Small, only first-stage init
System.img	Mounted at `/system`	Mounted as `/`, i.e. rootfs
Boot.img	Kernel + ramdisk	Same, but ramdisk is simpler
Recovery.img	Used for recovery boot only	In A/B: merged into boot.img

---

If you'd like, I can draw a visual diagram showing this evolution too. Want that?

28. Confusing

No problem — let’s break it down **in simple, clear points**, with no jargon.

✅ First, what is ramdisk?

- It's a **temporary file system** loaded into RAM when your phone first starts.
- It helps the device do the **very first steps** of booting.
- It **mounts important partitions** like:
  - `/system` → Android OS
  - `/vendor` → Hardware drivers
  - `/odm` → Board-specific configs

🧱 What is in boot.img?

- The `boot.img` file contains:
  1. **Kernel** → The heart of Android
  2. **Ramdisk** → Runs first to help mount everything

📦 What is in system.img?

- It has the full **Android OS** (system apps, services, etc.)
- After early boot, system.img is **mounted as** `/` (root directory)

🔄 What changed in Android 10 and 11?

🔹 Before Android 10:

- Ramdisk had **all boot logic and config**.
- system.img was mounted **inside** root (`/system`).

🔹 From Android 10 onward:

- Ramdisk became **minimal**.
  - It just runs **first_stage_init** (a small program).
  - It **mounts** the system, vendor, odm.
- `system.img` is now **mounted as root** `/` directly. This is called:

  System-as-root

📌 Two types of devices:

1. Non-A/B Devices

- `boot.img` has kernel + ramdisk
- `system.img` gets mounted as `/`
- After that, ramdisk is thrown away

2. A/B Devices (Seamless Updates)

- `boot.img` is also used for **recovery**
- It mounts from a folder called `/first_stage_ramdisk`
- Then everything proceeds the same

🔁 Summary Flow (Android 10+):

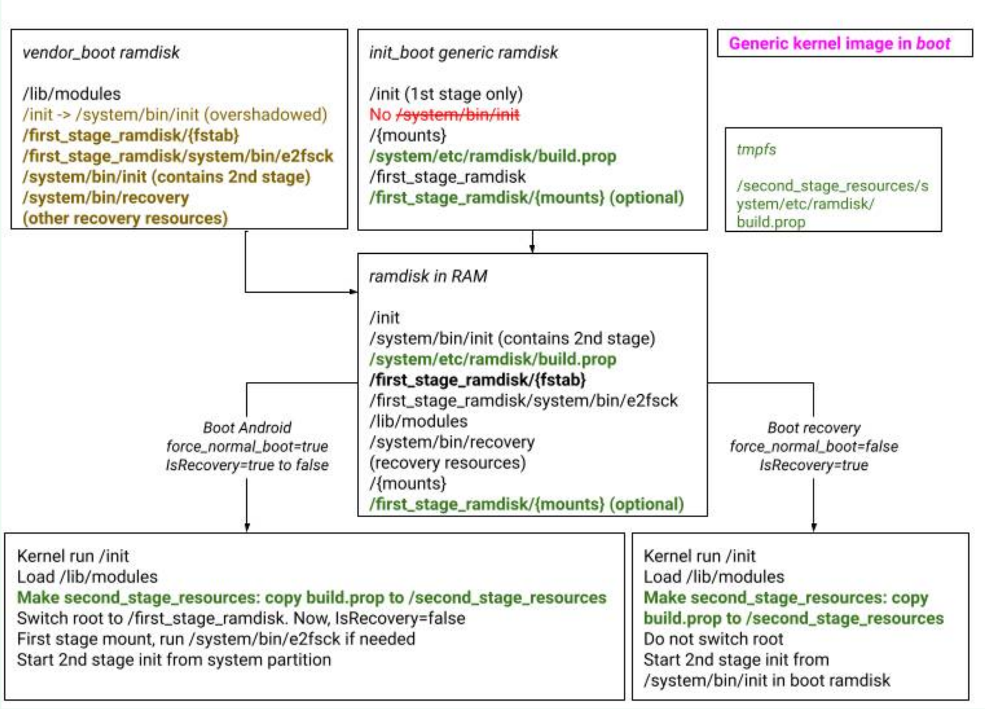
1. `boot.img` loads (contains kernel + small ramdisk)
2. **Ramdisk runs** `init` → **mounts** `/system`, `/vendor`, `/odm`
3. Root (`/`) becomes the **system.img**
4. Main Android OS starts from `/system/bin/init`

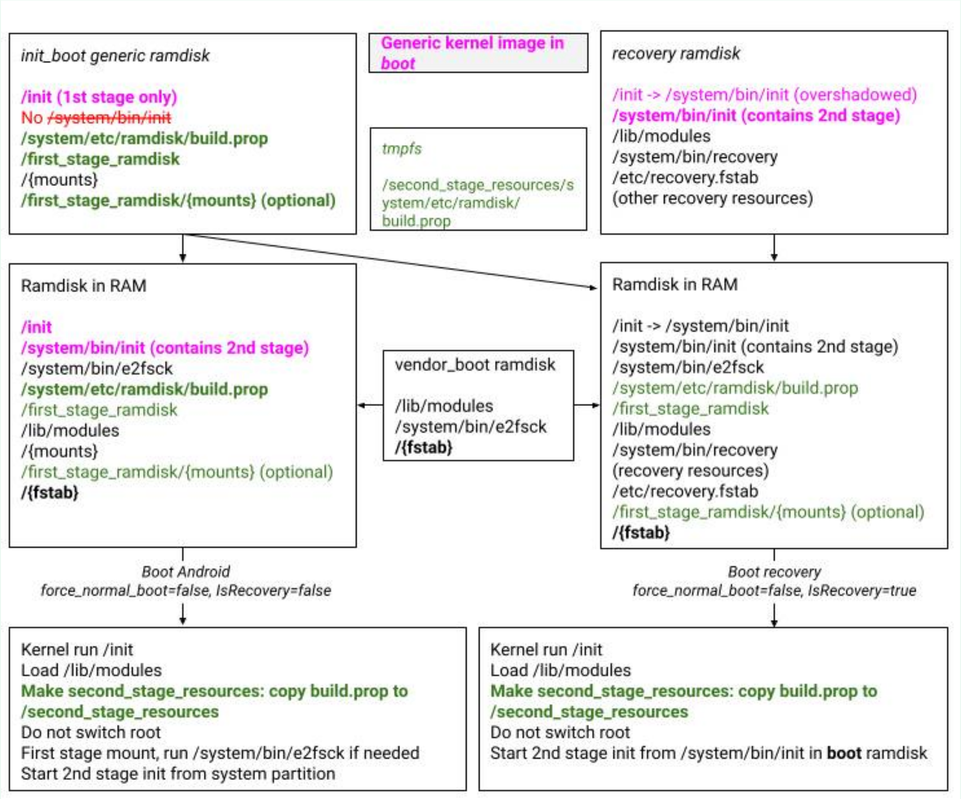
🚦Why was this changed?

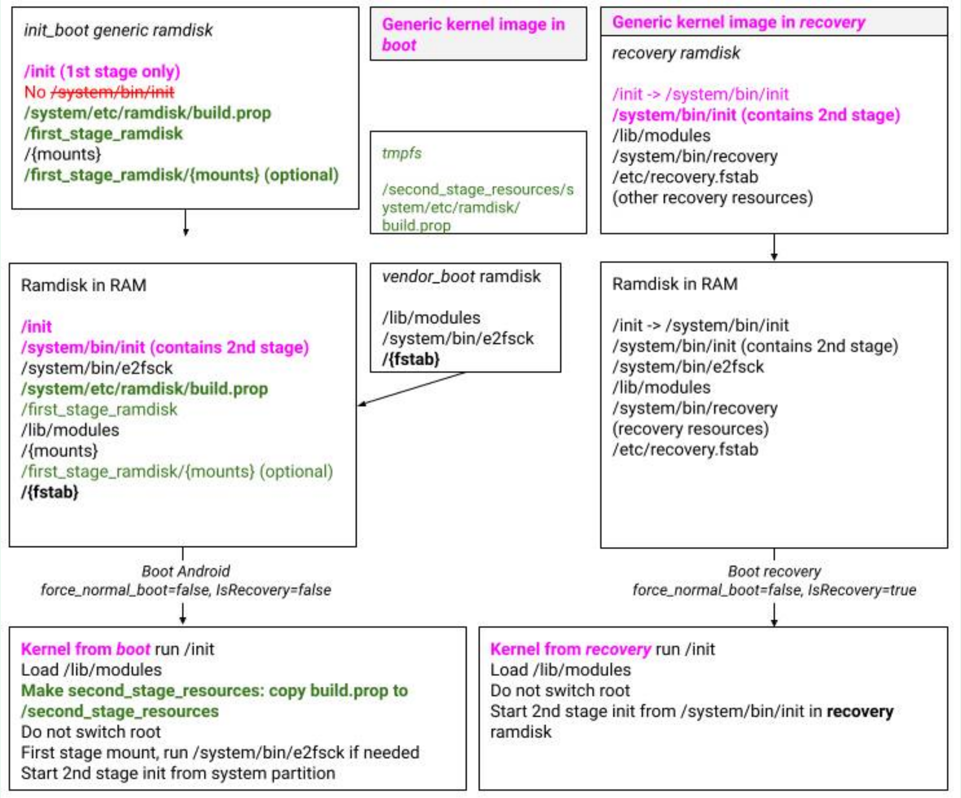
- To **speed up boot**
- To **support seamless A/B updates**
- To **cleanly separate early boot logic from main Android system**

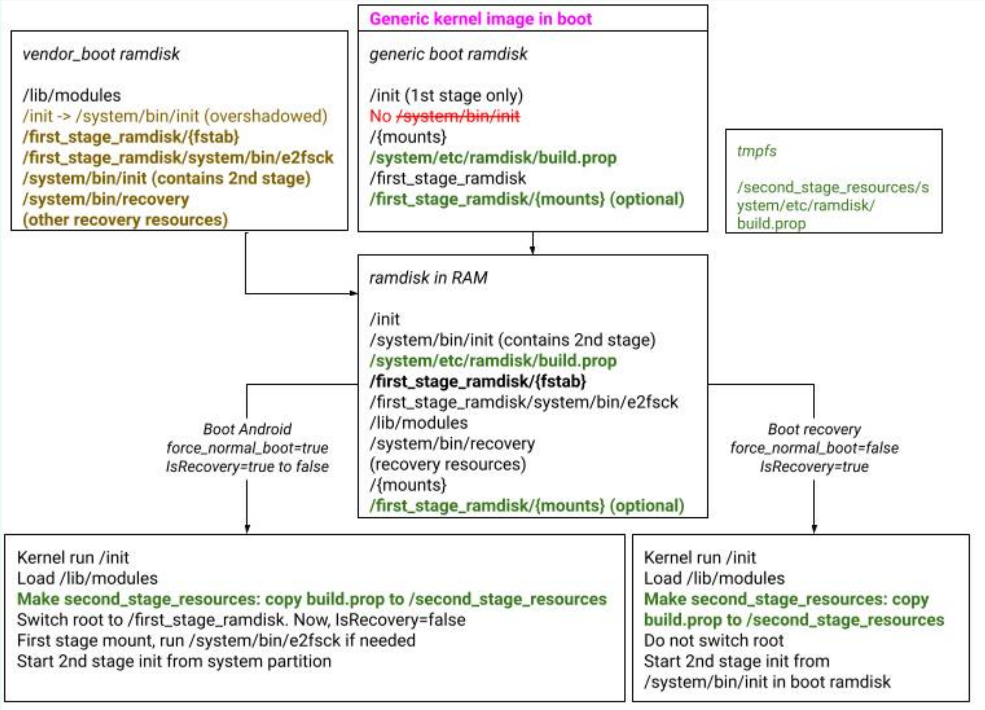
Want me to show you a **simple diagram** showing before and after Android 10?

29.


vendor_boot ramdisk
/lib/modules
/init -> /system/bin/init (overshadowed)
/first_stage_ramdisk/{fstab}
/first_stage_ramdisk/system/bin/e2fsck
/system/bin/init (contains 2nd stage)
/system/bin/recovery
(other recovery resources)
init_boot generic ramdisk
/init (1st stage only)
No ~~/system/bin/init~~
/{mounts}
/system/etc/ramdisk/build.prop
/first_stage_ramdisk
/first_stage_ramdisk/{mounts} (optional)
Generic kernel image in boot
tmpfs
/second_stage_resources/system/etc/ramdisk/build.prop
ramdisk in RAM
/init
/system/bin/init (contains 2nd stage)
/system/etc/ramdisk/build.prop
/first_stage_ramdisk/{fstab}
/first_stage_ramdisk/system/bin/e2fsck
/lib/modules
/system/bin/recovery
(recovery resources)
/{mounts}
/first_stage_ramdisk/{mounts} (optional)
Boot Android
force_normal_boot=true
IsRecovery=true to false
Boot recovery
force_normal_boot=false
IsRecovery=true
Kernel run /init
Load /lib/modules
Make second_stage_resources: copy build.prop to /second_stage_resources
Switch root to /first_stage_ramdisk. Now, IsRecovery=false
First stage mount, run /system/bin/e2fsck if needed
Start 2nd stage init from system partition
Kernel run /init
Load /lib/modules
Make second_stage_resources: copy build.prop to /second_stage_resources
Do not switch root
Start 2nd stage init from /system/bin/init in boot ramdisk


init_boot generic ramdisk
/init (1st stage only)
No ~~/system/bin/init~~
/system/etc/ramdisk/build.prop
/first_stage_ramdisk
/{mounts}
/first_stage_ramdisk/{mounts} (optional)
Generic kernel image in boot
tmpfs
/second_stage_resources/system/etc/ramdisk/build.prop
recovery ramdisk
/init -> /system/bin/init (overshadowed)
/system/bin/init (contains 2nd stage)
/lib/modules
/system/bin/recovery
/etc/recovery.fstab
(other recovery resources)
Ramdisk in RAM
/init
/system/bin/init (contains 2nd stage)
/system/bin/e2fsck
/system/etc/ramdisk/build.prop
/first_stage_ramdisk
/lib/modules
/{mounts}
/first_stage_ramdisk/{mounts} (optional)
/{fstab}
vendor_boot ramdisk
/lib/modules
/system/bin/e2fsck
/{fstab}
Ramdisk in RAM
/init -> /system/bin/init
/system/bin/init (contains 2nd stage)
/system/bin/e2fsck
/system/etc/ramdisk/build.prop
/first_stage_ramdisk
/lib/modules
/system/bin/recovery
(recovery resources)
/etc/recovery.fstab
/first_stage_ramdisk/{mounts} (optional)
/{fstab}
Boot Android
force_normal_boot=false, IsRecovery=false
Boot recovery
force_normal_boot=false, IsRecovery=true
Kernel run /init
Load /lib/modules
Make second_stage_resources: copy build.prop to /second_stage_resources
Do not switch root
First stage mount, run /system/bin/e2fsck if needed
Start 2nd stage init from system partition
Kernel run /init
Load /lib/modules
Make second_stage_resources: copy build.prop to /second_stage_resources
Do not switch root
Start 2nd stage init from /system/bin/init in boot ramdisk


init_boot generic ramdisk
/init (1st stage only)
No ~~/system/bin/init~~
/system/etc/ramdisk/build.prop
/first_stage_ramdisk
/{mounts}
/first_stage_ramdisk/{mounts} (optional)
Generic kernel image in boot
tmpfs
/second_stage_resources/system/etc/ramdisk/build.prop
Generic kernel image in recovery
recovery ramdisk
/init -> /system/bin/init
/system/bin/init (contains 2nd stage)
/lib/modules
/system/bin/recovery
/etc/recovery.fstab
(other recovery resources)
Ramdisk in RAM
/init
/system/bin/init (contains 2nd stage)
/system/bin/e2fsck
/system/etc/ramdisk/build.prop
/first_stage_ramdisk
/lib/modules
/{mounts}
/first_stage_ramdisk/{mounts} (optional)
/{fstab}
vendor_boot ramdisk
/lib/modules
/system/bin/e2fsck
/{fstab}
Ramdisk in RAM
/init -> /system/bin/init
/system/bin/init (contains 2nd stage)
/system/bin/e2fsck
/lib/modules
/system/bin/recovery
(recovery resources)
/etc/recovery.fstab
Boot Android
force_normal_boot=false, IsRecovery=false
Boot recovery
force_normal_boot=false, IsRecovery=true
Kernel from boot run /init
Load /lib/modules
Make second_stage_resources: copy build.prop to /second_stage_resources
Do not switch root
First stage mount, run /system/bin/e2fsck if needed
Start 2nd stage init from system partition
Kernel from recovery run /init
Load /lib/modules
Do not switch root
Start 2nd stage init from /system/bin/init in recovery ramdisk


Generic kernel image in boot
vendor_boot ramdisk
/lib/modules
/init -> /system/bin/init (overshadowed)
/first_stage_ramdisk/{fstab}
/first_stage_ramdisk/system/bin/e2fsck
/system/bin/init (contains 2nd stage)
/system/bin/recovery
(other recovery resources)
generic boot ramdisk
/init (1st stage only)
No ~~/system/bin/init~~
/{mounts}
/system/etc/ramdisk/build.prop
/first_stage_ramdisk
/first_stage_ramdisk/{mounts} (optional)
tmpfs
/second_stage_resources/system/etc/ramdisk/build.prop
ramdisk in RAM
/init
/system/bin/init (contains 2nd stage)
/system/etc/ramdisk/build.prop
/first_stage_ramdisk/{fstab}
/first_stage_ramdisk/system/bin/e2fsck
/lib/modules
/system/bin/recovery
(recovery resources)
/{mounts}
/first_stage_ramdisk/{mounts} (optional)
Boot Android
force_normal_boot=true
IsRecovery=true to false
Boot recovery
force_normal_boot=false
IsRecovery=true
Kernel run /init
Load /lib/modules
Make second_stage_resources: copy build.prop to /second_stage_resources
Switch root to /first_stage_ramdisk. Now, IsRecovery=false
First stage mount, run /system/bin/e2fsck if needed
Start 2nd stage init from system partition
Kernel run /init
Load /lib/modules
Make second_stage_resources: copy build.prop to /second_stage_resources
Do not switch root
Start 2nd stage init from /system/bin/init in boot ramdisk

Generic boot partition

bookmark_border
In Android 12, the generic boot image, referred to as Generic Kernel Image (GKI), contains the generic ramdisk and the GKI kernel.

For devices launching with Android 13, the generic ramdisk is removed from the boot image and placed in a separate init_boot image. This change leaves the boot image with only the GKI kernel.

For upgrading devices that continue to use Android 12 or older kernel versions, the generic ramdisk remains where it was with no requirement for a new init_boot image.

To build a generic ramdisk, move vendor-specific resources out of the ramdisk such that the generic ramdisk contains only first stage init and a property file that contains timestamp information.

On devices that:

Don't use a dedicated recovery partition, all recovery bits move from the generic ramdisk to vendor_boot ramdisk.

Do use a dedicated recovery partition, no change in the recovery ramdisk is needed because the recovery ramdisk is self-contained.

Architecture
The following diagrams illustrate the architecture for devices running Android 12 and higher. Device launching with Android 13 have a new init_boot image containing the generic ramdisk. Devices upgrading from Android 12 to Android 13 use the same architecture as they did with Android 12.

Launch with Android 13, no dedicated recovery
Launch/upgrade device, GKI, no dedicated recovery

Figure 1. Devices launching or upgrading to Android 13, with GKI, no dedicated recovery.

Launch with Android 13, dedicated and A/B recovery (dedicated ramdisk)
Launch/upgrade device, GKI, dedicated and A/B recovery

Figure 2. Devices launching or upgrading to Android 13, with GKI, dedicated and A/B recovery.

Refer to this figure if the device has recovery_a and recovery_b partitions.

Launch with Android 13, dedicated and non-A/B recovery (dedicated ramdisk)
Launch/upgrade device, GKI, dedicated and non-A/B recovery

Figure 3. Devices launching or upgrading to Android 13, with GKI, dedicated and non-A/B recovery.

Refer to this figure if the device has a partition named recovery without a slot suffix.

Launch or upgrade to Android 12, no dedicated recovery
Launch/upgrade device, GKI, no dedicated recovery

Figure 4. Devices launching or upgrading to Android 12, with GKI, no dedicated recovery.

Launch or upgrade to Android 12, dedicated and A/B recovery (dedicated ramdisk)
Launch/upgrade device, GKI, dedicated and A/B recovery

Figure 5. Devices launching or upgrading to Android 12, with GKI, dedicated and A/B recovery.

Refer to this figure if the device has recovery_a and recovery_b partitions.

Launch or upgrade to Android 12, dedicated and non-A/B recovery (dedicated ramdisk)
Launch/upgrade device, GKI, dedicated and non-A/B recovery

Figure 6. Devices launching or upgrading to Android 12, with GKI, dedicated and non-A/B recovery.

Refer to this figure if the device has a partition named recovery without a slot suffix.

Upgrade to Android 12, recovery-as-boot (recovery-as-ramdisk)
Launch/upgrade device, no GKI, recovery-as-boot

Figure 7. Devices upgrading to Android 12, no GKI, recovery-as-boot.

Upgrade to Android 12, dedicated recovery (dedicated ramdisk)
Launch/upgrade device, no GKI, dedicated recovery

Figure 8. Devices upgrading to Android 12, no GKI, dedicated recovery.

Boot images contents
The Android boot images contain the following.

init_boot image added for devices launching with Android 13

Header version V4
Generic ramdisk image
Generic boot image

Header version V3 or V4
A boot_signature for GKI boot.img certification (v4 only). The certified GKI boot.img isn't signed for verified boot. OEMs must still sign the prebuilt boot.img with a device-specific AVB key.
Generic cmdline (GENERIC_KERNEL_CMDLINE)
GKI kernel
Generic ramdisk image
Only included in boot images from Android 12 and earlier
vendor_boot image (for details, see Vendor Boot Partitions)

vendor_boot header
Device-specific cmdline (BOARD_KERNEL_CMDLINE)
vendor_boot ramdisk image
lib/modules
Recovery resources (if no dedicated recovery)
dtb image
recovery image

Header version V2
Device-specific cmdline for recovery, if necessary
For non-A/B recovery partition, contents of the header must be standalone; see Recovery Images. For example:
cmdline isn't concatenated to boot and vendor_boot cmdline.
Header specifies recovery DTBO, if necessary.
For A/B recovery partition, contents can be concatenated or inferred from boot and vendor_boot. For example:
cmdline is concatenated to boot and vendor_boot cmdline.
DTBO can be inferred from vendor_boot header.
recovery ramdisk image
Recovery resources
For non-A/B recovery partition, contents of the ramdisk must be standalone; see Recovery Images. For example:
lib/modules must contain all kernel modules required to boot recovery mode
The recovery ramdisk must contain init.
For A/B recovery partition, the recovery ramdisk is prepended to the generic and vendor_boot ramdisk, hence it doesn't need to be standalone. For example:
lib/modules might contain only additional kernel modules required to boot recovery mode besides kernel modules in vendor_boot ramdisk.
The symlink at /init might exist, but it is overshadowed by the the first-stage /init binary in boot image.
Generic ramdisk image contents
The generic ramdisk contains the following components.

init
system/etc/ramdisk/build.prop
ro.PRODUCT.bootimg.* build props
Empty directories for mount points: debug_ramdisk/, mnt/, dev/, sys/, proc/, metadata/
first_stage_ramdisk/
Duplicated empty directories for mount points: debug_ramdisk/, mnt/, dev/, sys/, proc/, metadata/
Boot image integration
Build flags control how init_boot, boot, recovery, and vendor_boot images are built. The value of a boolean board variable must be the string true or be empty (which is the default).

TARGET_NO_KERNEL. This variable indicates if the build uses a prebuilt boot image. If this variable is set to true, then set BOARD_PREBUILT_BOOTIMAGE to the location of the prebuilt boot image (BOARD_PREBUILT_BOOTIMAGE:= device/${company}/${board}/boot.img)

BOARD_USES_RECOVERY_AS_BOOT. This variable indicates whether the device uses the recovery image as the boot image. When using GKI, this variable is empty and recovery resources should be moved to vendor_boot.

BOARD_USES_GENERIC_KERNEL_IMAGE. This variable indicates that the board uses GKI. This variable doesn't affect sysprops or PRODUCT_PACKAGES.

This is the board-level GKI switch; all of the following variables are restricted by this variable.

BOARD_MOVE_RECOVERY_RESOURCES_TO_VENDOR_BOOT. This variable controls whether ramdisk recovery resources are built to vendor_boot.

When set to true, recovery resources are built to vendor-ramdisk/ only and aren't built to recovery/root/.

When empty, recovery resources are built to recovery/root/ only and aren't built to vendor-ramdisk/.

BOARD_MOVE_GSI_AVB_KEYS_TO_VENDOR_BOOT. This variable controls whether GSI AVB keys are built to vendor_boot.

When set to true, if BOARD_MOVE_RECOVERY_RESOURCES_TO_VENDOR_BOOT:

Is set, GSI AVB keys are built to $ANDROID_PRODUCT_OUT/vendor-ramdisk/first_stage_ramdisk/avb.

Is unset, GSI AVB keys are built to $ANDROID_PRODUCT_OUT/vendor-ramdisk/avb.

When empty, if BOARD_RECOVERY_AS_ROOT:

Is set, GSI AVB keys are built to $ANDROID_PRODUCT_OUT/recovery/root/first_stage_ramdisk/avb.

Is unset, GSI AVB keys are built to $ANDROID_PRODUCT_OUT/ramdisk/avb.

BOARD_EXCLUDE_KERNEL_FROM_RECOVERY_IMAGE. This variable controls whether the recovery image contains a kernel or not. Devices launching with Android 12 and using A/B recovery partition must set this variable to true. Devices launching with Android 12 and using non-A/B must set this variable to false to keep the recovery image self-contained.

BOARD_COPY_BOOT_IMAGE_TO_TARGET_FILES. This variable controls whether $OUT/boot*.img is copied to IMAGES/ under target files.

aosp_arm64 must set this variable to true.

Other devices must leave this variable empty.

BOARD_INIT_BOOT_IMAGE_PARTITION_SIZE. This variable controls whether init_boot.img is generated and sets the size. When set, the generic ramdisk is added to the init_boot.img instead of boot.img and requires the BOARD_AVB_INIT_BOOT* variables to be set for chained vbmeta.

Allowed combinations

Component or variable	Upgrade device without recovery partition	Upgrade device with recovery partition	Launch device without recovery partition	Launch device with A/B recovery partition	Launch device with non-A/B recovery partition	aosp_arm64
Contains boot	yes	yes	yes	yes	yes	yes
Contains init_boot (Android 13)	no	no	yes	yes	yes	yes
Contains vendor_boot	optional	optional	yes	yes	yes	no
Contains recovery	no	yes	no	yes	yes	no
BOARD_USES_RECOVERY_AS_BOOT	true	empty	empty	empty	empty	empty
BOARD_USES_GENERIC_KERNEL_IMAGE	empty	empty	true	true	true	true
PRODUCT_BUILD_RECOVERY_IMAGE	empty	true or empty	empty	true or empty	true or empty	empty
BOARD_RECOVERYIMAGE_PARTITION_SIZE	empty	> 0	empty	> 0	> 0	empty
BOARD_MOVE_RECOVERY_RESOURCES_TO_VENDOR_BOOT	empty	empty	true	empty	empty	empty
BOARD_MOVE_GSI_AVB_KEYS_TO_VENDOR_BOOT	empty	empty	true	true	true	empty
BOARD_EXCLUDE_KERNEL_FROM_RECOVERY_IMAGE	empty	empty	empty	true	empty	empty
BOARD_COPY_BOOT_IMAGE_TO_TARGET_FILES	empty	empty	empty	empty	empty	true

Devices with a dedicated recovery partition can set PRODUCT_BUILD_RECOVERY_IMAGE to true or empty. For these devices, if BOARD_RECOVERYIMAGE_PARTITION_SIZE is set, a recovery image is built.

Enable chained vbmeta for boot
Chained vbmeta must be enabled for the boot and init_boot images. Specify the following:

```
BOARD_AVB_BOOT_KEY_PATH := external/avb/test/data/testkey_rsa4096.pem
BOARD_AVB_BOOT_ALGORITHM := SHA256_RSA4096
BOARD_AVB_BOOT_ROLLBACK_INDEX := $(PLATFORM_SECURITY_PATCH_TIMESTAMP)
BOARD_AVB_BOOT_ROLLBACK_INDEX_LOCATION := 2

BOARD_AVB_INIT_BOOT_KEY_PATH := external/avb/test/data/testkey_rsa2048.pem
BOARD_AVB_INIT_BOOT_ALGORITHM := SHA256_RSA2048
BOARD_AVB_INIT_BOOT_ROLLBACK_INDEX := $(PLATFORM_SECURITY_PATCH_TIMESTAMP)
BOARD_AVB_INIT_BOOT_ROLLBACK_INDEX_LOCATION := 3
```

For an example, refer to this change.

System-as-root
System-as-root isn't supported for devices that use GKI. On such devices, BOARD_BUILD_SYSTEM_ROOT_IMAGE must be empty. System-as-root also isn't supported for devices that use dynamic partitions.

Note: In theory, a device that uses system-as-root and a dedicated recovery partition could have a system image that includes a first_stage_ramdisk directory. In practice however, that directory doesn't have any defined file_contexts.
Product configurations
Devices that use the generic ramdisk must install a list of files that are allowed to be installed to the ramdisk. To do so, specify the following in device.mk:

```
$(call inherit-product, $(SRC_TARGET_DIR)/product/generic_ramdisk.mk)
```

The generic_ramdisk.mk file also prevents other makefiles from accidentally installing other files to the ramdisk (move such files to vendor_ramdisk instead).

Set up devices
Setup instructions differ between devices launching with Android 13, upgrading to Android 12, and launching with Android 12. Android 13, are setup similar to how they were with Android 12

Devices Upgrading to Android 12:

Can preserve the value of BOARD_USES_RECOVERY_AS_BOOT. If they do so, they're using legacy configs and new build variables must be empty. If such devices:

Set BOARD_USES_RECOVERY_AS_BOOT to true, the architecture is as shown in Figure 3.

Set BOARD_USES_RECOVERY_AS_BOOT to empty, the architecture is as shown Figure 4.

Can set BOARD_USES_RECOVERY_AS_BOOT to empty. If they do so, they're using new configurations. If such devices:

Don't use a dedicated recovery partition, the architecture is as shown in Figure 1 and the device setup option is Option 1.

Use a dedicated recovery partition, the architecture is as shown in Figure 2a or Figure 2b and the device setup option is Option 2a or Option 2b.

Devices launching with Android 12 must set BOARD_USES_RECOVERY_AS_BOOT to empty and use new configurations. If such devices:

Don't use a dedicated recovery partition, the architecture is as shown in Figure 1 and the device setup option is Option 1.

Use a dedicated recovery partition, the architecture is as shown in Figure 2a or Figure 2b and the device setup option is Option 2a or Option 2b.

Because aosp_arm64 builds only GKI (and not vendor_boot or recovery), it isn't a complete target. For aosp_arm64build configurations, refer to generic_arm64.

Option 1: No dedicated recovery partition
Devices without a recovery partition contain the generic boot image in the boot partition. The vendor_boot ramdisk contains all recovery resources, including lib/modules (with vendor kernel modules). On such devices, the product configuration inherits from generic_ramdisk.mk.

Set BOARD values
Set the following values:

```
BOARD_USES_RECOVERY_AS_BOOT :=
BOARD_USES_GENERIC_KERNEL_IMAGE := true
BOARD_MOVE_RECOVERY_RESOURCES_TO_VENDOR_BOOT := true
BOARD_EXCLUDE_KERNEL_FROM_RECOVERY_IMAGE :=
BOARD_MOVE_GSI_AVB_KEYS_TO_VENDOR_BOOT := true
```

Init binaries and symlinks
The vendor_boot ramdisk can contain an /init to /system/bin/init symlink, and init_second_stage.recovery at /system/bin/init. However, because the generic ramdisk is concatenated after the vendor_boot ramdisk, the /init symlink is overwritten. When the device boots into recovery, the /system/bin/init binary is needed to support second stage init. The contents of vendor_boot + generic ramdisks are as follows:

/init (from generic ramdisk, built from init_first_stage)
/system/bin/init (from vendor_ramdisk, built from init_second_stage.recovery)
Move fstab files
Move any fstab files that were installed to the generic ramdisk to vendor_ramdisk. For an example, refer to this change.

Install modules
You can install device-specific modules to vendor_ramdisk (skip this step if you don't have any device-specific modules to install).

Use the vendor_ramdisk variant of the module when the module installs to the /first_stage_ramdisk. This module should be available after init switches root into /first_stage_ramdisk but before init switches root into /system. For examples, see Metadata checksums and Virtual A/B compression.

Use the recovery variant of the module when the module installs to /. This module should be available before init switches root into /first_stage_ramdisk. For details on installing modules to /, see First stage console.

First stage console
Because the first stage console starts before init switches root into /first_stage_ramdisk, you need to install the recovery variant of modules. By default, both module variants are installed to build/make/target/product/base_vendor.mk, so if the device makefile inherits from that file you don't need to explicitly install the recovery variant.

To explicitly install the recovery modules, use the following.

```
PRODUCT_PACKAGES += \
    linker.recovery \
    shell_and_utilities_recovery \
```

This ensures that the linker, sh, and toybox install to $ANDROID_PRODUCT_OUT/recovery/root/system/bin, which then installs to /system/bin under the vendor_ramdisk.

To add modules needed for the first stage console (for example, adbd), use the following.

```
PRODUCT_PACKAGES += adbd.recovery
```

This ensures that the specified modules install to $ANDROID_PRODUCT_OUT/recovery/root/system/bin, which then installs to /system/bin under the vendor_ramdisk.

Important: Don't install the vendor_ramdisk variant! That variant installs to /first_stage_ramdisk/system/bin under the vendor_ramdisk.
Metadata checksums
To support metadata checksums during first stage mount, devices that don't support GKI install the ramdisk variant of the following modules. To add support for GKI, move the modules to $ANDROID_PRODUCT_OUT/vendor-ramdisk/first_stage_ramdisk/system/bin:

```
PRODUCT_PACKAGES += \
    linker.vendor_ramdisk \
    resize2fs.vendor_ramdisk \
    tune2fs.vendor_ramdisk \
```

For an example, refer to this changelist.

Virtual A/B compression
To support virtual A/B compression, snapuserd must be installed to vendor_ramdisk. The device should inherit from virtual_ab_ota/compression.mk, which installs the vendor_ramdisk variant of snapuserd.

Changes to the boot process
The process of booting into recovery or into Android doesn't change, with the following exception:

Ramdisk build.prop moves into /second_stage_resources so that second stage init can read the build timestamp of boot.
Because resources move from generic ramdisk to vendor_boot ramdisk, the result of concatenating generic ramdisk to vendor_boot ramdisk doesn't change.

Make e2fsck available
The device makefiles can inherit from:

virtual_ab_ota/launch_with_vendor_ramdisk.mk if the device supports virtual A/B but not compression.

virtual_ab_ota/compression.mk if the device supports virtual A/B compression.

The product makefiles install $ANDROID_PRODUCT_OUT/vendor-ramdisk/first_stage_ramdisk/system/bin/e2fsck. At runtime, the first stage init switches root into /first_stage_ramdisk then executes /system/bin/e2fsck.

Option 2a: Dedicated and A/B recovery partition
Use this option for devices with A/B recovery partitions; that is, the device has a recovery_a and recovery_b partition. Such devices include A/B and Virtual A/B devices of which the recovery partition is updateable, with the following configuration:

```
AB_OTA_PARTITIONS += recovery
```

The vendor_boot ramdisk contains the vendor bits of the ramdisk and vendor kernel modules, including the following:

Device-specific fstab files

lib/modules (includes vendor kernel modules)

The recovery ramdisk contains all recovery resources. On such devices, the product configuration inherits from generic_ramdisk.mk.

Set BOARD values
Set the following values for devices with A/B recovery partition:

```
BOARD_USES_RECOVERY_AS_BOOT :=
BOARD_USES_GENERIC_KERNEL_IMAGE := true
BOARD_MOVE_RECOVERY_RESOURCES_TO_VENDOR_BOOT :=
BOARD_EXCLUDE_KERNEL_FROM_RECOVERY_IMAGE := true
BOARD_MOVE_GSI_AVB_KEYS_TO_VENDOR_BOOT := true
```

Init binaries and symlinks
The recovery ramdisk can contain an /init -> /system/bin/init symlink, and init_second_stage.recovery at /system/bin/init. However, because the boot ramdisk is concatenated after the recovery ramdisk, the /init symlink is overwritten. When the device boots into recovery mode, the /system/bin/init binary is needed to support second stage init.

When the device boots into recovery, the contents of recovery + vendor_boot + generic ramdisks are as follows:

/init (from ramdisk, built from init_first_stage)
/system/bin/init (from recovery ramdisk, built from init_second_stage.recovery, and executed from /init)
When the device boots into Android, the contents of vendor_boot + generic ramdisks are as follows:

/init (from generic ramdisk, built from init_first_stage)
Move fstab files
Move any fstab files that were installed to the generic ramdisk to the vendor_ramdisk. For an example, refer to this change.

Install modules
Optionally, you can install device-specific modules to vendor_ramdisk (skip this step if you don't have any device-specific modules to install). Init doesn't switch root. The vendor_ramdisk variant of modules installs to the root of vendor_ramdisk. For examples on installing modules to vendor_ramdisk, see First stage console, Metadata checksums, and Virtual A/B compression.

First stage console
To install the vendor_ramdisk variant of the modules, use the following:

```
PRODUCT_PACKAGES += \
    linker.vendor_ramdisk \
    shell_and_utilities_vendor_ramdisk \
```

This ensures that the linker, sh, and toybox install to $ANDROID_PRODUCT_OUT/vendor-ramdisk/system/bin, which then installs to /system/bin under the vendor_ramdisk.

To add modules needed for the first stage console (for example, adbd), enable the vendor_ramdisk variant of these modules by uploading relevant patches to AOSP, then use the following,

```
PRODUCT_PACKAGES += adbd.vendor_ramdisk
```

This ensures that the specified modules install to $ANDROID_PRODUCT_OUT/vendor-ramdisk/system/bin. If the vendor_boot ramdisk is loaded in recovery mode, the module is also available in recovery. If the vendor_boot ramdisk isn't loaded in recovery mode, the device can optionally install adbd.recovery as well.

Metadata checksums
To support metadata checksums during first stage mount, devices that don't support GKI install the ramdisk variant of the following modules. To add support for GKI, move the modules to $ANDROID_PRODUCT_OUT/vendor-ramdisk/system/bin:

```
PRODUCT_PACKAGES += \
    linker.vendor_ramdisk \
    resize2fs.vendor_ramdisk \
    tune2fs.vendor_ramdisk \
```

For an example, refer to this changelist.

Virtual A/B compression
To support Virtual A/B compression, snapuserd must be installed to vendor_ramdisk. The device should inherit from virtual_ab_ota/compression.mk, which installs the vendor_ramdisk variant of snapuserd.

Changes to the boot process
When booting into Android, the boot process doesn't change. The vendor_boot + generic ramdisk is similar to the existing boot process, except that fstab loads from vendor_boot. Because system/bin/recovery doesn't exist, first_stage_init handles it as a normal boot.

When booting into recovery mode, the boot process changes. The recovery + vendor_boot + generic ramdisk is similar to the existing recovery process, but the kernel is loaded from the boot image instead of from the recovery image. The boot process for recovery mode is as follows.

Bootloader starts, then does the following:

Pushes recovery + vendor_boot + generic ramdisk to /. (If the OEM duplicates kernel modules in recovery ramdisk by adding them to BOARD_RECOVERY_KERNEL_MODULES), vendor_boot is optional.)
Runs the kernel from the boot partition.
Kernel mounts ramdisk to / then executes /init from the generic ramdisk.

First stage init starts, then does the following:

Sets IsRecoveryMode() == true and ForceNormalBoot() == false.
Loads vendor kernel modules from /lib/modules.
Calls DoFirstStageMount() but skips mounting because IsRecoveryMode() == true. (The device doesn't free ramdisk (because / is still the same) but does call SetInitAvbVersionInRecovery().)
Starts second stage init from /system/bin/init from recovery ramdisk.
Make e2fsck available
The device makefiles can inherit from:

virtual_ab_ota/launch_with_vendor_ramdisk.mk if the device supports virtual A/B but not compression.

virtual_ab_ota/compression.mk if the device supports virtual A/B compression.

The product makefiles install $ANDROID_PRODUCT_OUT/vendor-ramdisk/system/bin/e2fsck. At runtime, the first stage init executes /system/bin/e2fsck.

Option 2b: Dedicated and non-A/B recovery partition
Use this option for devices with a non-A/B recovery partition; that is, the device has a partition named recovery without a slot suffix. Such devices include:

non-A/B devices;
A/B and Virtual A/B devices, of which the recovery partition isn't updateable. (This is unusual.)
The vendor_boot ramdisk contains the vendor bits of the ramdisk and vendor kernel modules, including the following:

Device-specific fstab files
lib/modules (includes vendor kernel modules)
The recovery image must be self-contained. It must contain all required resources to boot the recovery mode, including:

The kernel image
The DTBO image
Kernel modules in lib/modules
First-stage init as a symlink /init -> /system/bin/init
Second-stage init binary /system/bin/init
Device-specific fstab files
All other recovery resources, including the recovery binary
On such devices, the product configuration inherits from generic_ramdisk.mk.

Set BOARD values
Set the following values for non-A/B devices:

```
BOARD_USES_RECOVERY_AS_BOOT :=
BOARD_USES_GENERIC_KERNEL_IMAGE := true
BOARD_MOVE_RECOVERY_RESOURCES_TO_VENDOR_BOOT :=
BOARD_EXCLUDE_KERNEL_FROM_RECOVERY_IMAGE :=
BOARD_MOVE_GSI_AVB_KEYS_TO_VENDOR_BOOT := true
```

Init binaries and symlinks

The recovery ramdisk must contain an /init -> /system/bin/init symlink, and init_second_stage.recovery at /system/bin/init. When the device boots into recovery mode, the /system/bin/init binary is needed to support both first stage and second stage init.

When the device boots into recovery, the contents of recovery ramdisks are as follows:

/init -> /system/bin/init (from recovery ramdisk)

/system/bin/init (from recovery ramdisk, built from init_second_stage.recovery, and executed from /init)

When the device boots into Android, the contents of vendor_boot + generic ramdisks are as follows:

/init (from ramdisk, built from init_first_stage)

Move fstab files

Move any fstab files that were installed to the generic ramdisk to the vendor_ramdisk and recovery ramdisk. For an example, refer to this change.

Install modules

You can install device-specific modules to vendor_ramdisk and recovery ramdisk (skip this step if you don't have any device-specific modules to install). init doesn't switch root. The vendor_ramdisk variant of modules installs to the root of vendor_ramdisk. The recovery variant of modules installs to the root of recovery ramdisk. For examples on installing modules to vendor_ramdisk and recovery ramdisk, se First stage console and Metadata checksums.

First stage console

To install the vendor_ramdisk variant of the modules, use the following:

```
PRODUCT_PACKAGES += \
    linker.vendor_ramdisk \
    shell_and_utilities_vendor_ramdisk \
```

This ensures that the linker, sh, and toybox install to $ANDROID_PRODUCT_OUT/vendor-ramdisk/system/bin, which then installs to /system/bin under the vendor_ramdisk.

To add modules needed for the first stage console (for example, adbd), enable the vendor_ramdisk variant of these modules by uploading relevant patches to AOSP, then use the following,

```
PRODUCT_PACKAGES += adbd.vendor_ramdisk
```

This ensures that the specified modules install to $ANDROID_PRODUCT_OUT/vendor-ramdisk/system/bin.

To install the recovery variant of the modules, replace vendor_ramdisk with recovery:

```
PRODUCT_PACKAGES += \
    linker.recovery \
    shell_and_utilities_recovery \
    adbd.recovery \
```

Metadata checksums

To support metadata checksums during first stage mount, devices that don't support GKI install the ramdisk variant of the following modules. To add support for GKI, move the modules to $ANDROID_PRODUCT_OUT/vendor-ramdisk/system/bin:

```
PRODUCT_PACKAGES += \
    linker.vendor_ramdisk \
    resize2fs.vendor_ramdisk \
    tune2fs.vendor_ramdisk \
```

To support metadata checksums during first stage mount in recovery, enable the recovery variant of these modules and install them as well.

Changes to the boot process

When booting into Android, the boot process doesn't change. The vendor_boot + generic ramdisk is similar to the existing boot process, except that fstab loads from vendor_boot. Because system/bin/recovery doesn't exist, first_stage_init handles it as a normal boot.

When booting into recovery mode, the boot process doesn't change. The recovery ramdisk is loaded in the same way as the existing recovery process. The kernel is loaded from the recovery image. The boot process for recovery mode is as follows.

Bootloader starts, then does the following:

Pushes recovery ramdisk to /.

Runs the kernel from the recovery partition.
Kernel mounts ramdisk to / then executes /init, which is a symlink to /system/bin/init from the recovery ramdisk.

First stage init starts, then does the following:

Sets IsRecoveryMode() == true and ForceNormalBoot() == false.
Loads vendor kernel modules from /lib/modules.
Calls DoFirstStageMount() but skips mounting because IsRecoveryMode() == true. (The device doesn't free ramdisk (because / is still the same) but does call SetInitAvbVersionInRecovery().)
Starts second stage init from /system/bin/init from recovery ramdisk.
Boot image timestamps
The following code is an example boot image timestamp file:

```
####################################
# from generate-common-build-props
# These properties identify this partition image.
####################################
ro.product.bootimage.brand=Android
ro.product.bootimage.device=generic_arm64
ro.product.bootimage.manufacturer=unknown
ro.product.bootimage.model=AOSP on ARM64
ro.product.bootimage.name=aosp_arm64
ro.bootimage.build.date=Mon Nov 16 22:46:27 UTC 2020
ro.bootimage.build.date.utc=1605566787
ro.bootimage.build.fingerprint=Android/aosp_arm64/generic_arm64:S/MASTER/6976199:userdebug/test-keys
ro.bootimage.build.id=MASTER
ro.bootimage.build.tags=test-keys
ro.bootimage.build.type=userdebug
ro.bootimage.build.version.incremental=6976199
ro.bootimage.build.version.release=11
ro.bootimage.build.version.release_or_codename=S
ro.bootimage.build.version.sdk=30
# Auto-added by post_process_props.py
persist.sys.usb.config=none
# end of file
```

At build time, a system/etc/ramdisk/build.prop file is added to the generic ramdisk. This file contains timestamp information of the build.

At runtime, first stage init copies files from the ramdisk to tmpfs before freeing the ramdisk so that second stage init can read this file to set boot image timestamps properties.

Thanks for sharing all the images and text. Since the boot process with `vendor_boot`, `init_boot`, and `recovery` ramdisks can be very confusing, I'll now **simplify** and **explain the key concepts** clearly using your diagrams and the detailed document you shared.

🔧 Why Android Split the Ramdisk?

In **Android 12+ (especially 13)**, Android moved toward a **modular boot** approach called **GKI** (Generic Kernel Image). To support this:

- `boot.img` **contains only the kernel**
- `init_boot.img` **contains the generic ramdisk** (first stage only)
- `vendor_boot.img` **contains vendor-specific ramdisk resources** (second stage init, fstab, e2fsck, recovery binaries)
- `recovery.img` is used only if the device has a **dedicated recovery partition**.

🧠 Basic Concepts (in simple words)

Term	Simple Meaning
First stage init	Minimal init in `init_boot.img`, mounts partitions like system/vendor.
Second stage init	Full init in `/system/bin/init`, continues booting Android or Recovery.
init_boot.img	New (Android 13+) image with generic ramdisk for first-stage init.
vendor_boot.img	Vendor-specific ramdisk resources and second-stage init.
recovery.img	Used **only** if device has a recovery partition. Contains full recovery init and binaries.

Term	Simple Meaning
/init	Executed by the kernel. From first-stage (generic), and sometimes symlinked to second-stage.

✅ Boot Flow Scenarios (based on images)

🔵 Case 1: Android Boot (No Recovery Partition)

(Refer to `ramdisk-split-1a.png`)

- `boot.img` → GKI kernel
- `init_boot.img` → `/init` (first-stage init only)
- `vendor_boot.img` →
  - `/system/bin/init` (second-stage init)
  - `/system/bin/e2fsck`
  - `/first_stage_ramdisk/fstab`
  - `recovery resources`

Steps:

1. Kernel runs `/init` from `init_boot.img` (first-stage init)
2. Copies `build.prop` to `/second_stage_resources`
3. Mounts system/vendor using `fstab`
4. Executes `/system/bin/init` from `vendor_boot.img` (second stage)

🔴 Case 2a: Recovery Boot (A/B Device)

(Refer to `ramdisk-split-2a.png`)

- Kernel from `boot.img`
- Ramdisks: recovery + vendor_boot + generic
- `/init` from generic ramdisk
- `/system/bin/init` from recovery ramdisk

Steps:

1. Kernel executes `/init` (first-stage)
2. `IsRecovery = true`, `force_normal_boot = false`
3. Starts second-stage init from `/system/bin/init` in **recovery ramdisk**

🟣 Case 2b: Recovery Boot (Non-A/B Device)

(Refer to `ramdisk-split-2b.png`)

- Kernel and ramdisk from `recovery.img`
- `/init → /system/bin/init` (symlink)
- Self-contained recovery image
  - All kernel modules
  - Recovery fstab
  - No vendor_boot required

Steps:

1. Bootloader runs kernel from `recovery.img`
2. `/init` (symlink) runs `/system/bin/init` in recovery

3. **`IsRecovery = true`**, starts recovery mode directly

🔄 Summary Table: Boot Flow by Scenario

Boot Type	Image Used	Kernel From	Init Used	Second Stage Init From
Android	`boot.img`, `init_boot.img`, `vendor_boot.img`	`boot.img`	`/init` from `init_boot`	`/system/bin/init` from `vendor_boot`
Recovery (A/B)	`boot.img`, `recovery ramdisk`, `vendor_boot.img`	`boot.img`	`/init` from generic	`/system/bin/init` from **recovery**
Recovery (non-A/B)	`recovery.img` only	`recovery.img`	`/init → /system/bin/init`	`/system/bin/init` from **recovery**

⚙️ Key Build Variables (Device.mk / BoardConfig.mk)

Variable	Meaning
`BOARD_USES_GENERIC_KERNEL_IMAGE := true`	Enables GKI support
`BOARD_MOVE_RECOVERY_RESOURCES_TO_VENDOR_BOOT := true`	Moves recovery resources to `vendor_boot`
`BOARD_USES_RECOVERY_AS_BOOT :=`	Empty = not using recovery-as-boot
`BOARD_EXCLUDE_KERNEL_FROM_RECOVERY_IMAGE := true`	For A/B recovery, recovery.img doesn't contain kernel

🧪 What You Can Do Next (To Practice or Debug)

- Use `unpack_bootimg.py` to inspect each image.
- Run `fastboot boot recovery.img` to test recovery flow.
- Explore `/init.rc` in generic and recovery builds.
- Read `fstab*` in vendor_boot to understand first-stage mount.

Would you like a **custom diagram or table** summarizing all flows for **launching with or without recovery partition** in Android 12/13?

30. so there is different boot flow in recovery mode?

Yes, **recovery mode has a different boot flow** compared to normal Android boot. Let's break it down in **simple terms** with visuals in mind.

✅ Basic Boot Flow Difference: Normal vs Recovery

Aspect	🟢 Normal Android Boot	🔴 Recovery Boot
Goal	Boot into full Android OS	Boot into recovery UI (e.g., for factory reset, sideload)
Kernel	From **`boot.img`**	From **`boot.img`** (A/B) or **`recovery.img`** (non-A/B)
Ramdisk	Split (init_boot + vendor_boot)	Includes recovery ramdisk + vendor_boot
`/init` (first stage)	From **`init_boot.img`**	From **`init_boot.img`** (or from recovery ramdisk)
Second stage init	From **`/system/bin/init`** in **`vendor_boot`**	From **`/system/bin/init`** in **recovery ramdisk**
`IsRecoveryMode()` flag	**`false`**	**`true`** (set by kernel cmdline or bootloader)
Starts Android UI?	Yes	No (starts **`recovery`** UI process)

🧭 Recovery Boot: Key Differences

🔹 A/B Devices (e.g., Pixel)

- Have **no separate `recovery.img` partition**
- Kernel is loaded from **`boot.img`**
- Recovery **ramdisk is included** in **`vendor_boot.img`**
- **`init`** runs from **`init_boot.img`**
- If **`is_recovery == true`**, then **`/system/bin/init`** is taken from **recovery ramdisk** (not vendor).

🔸 Non-A/B Devices

- Use a dedicated **`recovery.img`**
- Both **kernel and ramdisk** come from **`recovery.img`**
- **`init`** is either a real binary or symlink to **`/system/bin/init`** (in ramdisk)
- Entire recovery boot is **self-contained**

🧠 What Triggers Recovery Boot?

- Bootloader passes special **`cmdline`**:

  ```ini
  androidboot.bootreason=reboot,recovery
  androidboot.slot_suffix=_a
  androidboot.mode=recovery
  ```

- Then **`init`** detects this, and:
  - Loads **`init.recovery.*.rc`**
  - Starts **`recovery`** process (e.g., **`/sbin/recovery`** or **`/system/bin/recovery`**)

🖼️ Visual Flow (Simplified)

```swift
Recovery Boot (A/B):

boot.img ──▶ Kernel
             │
```

```
           ▼
init_boot.img ──▶ /init  (first stage)
           │
vendor_boot.img
  ├─ recovery ramdisk
  └─ /system/bin/init  ← (second stage)
           │
           ▼
IsRecovery → true ──▶ Launch recovery.rc → UI
```

```swift
Recovery Boot (non-A/B):

recovery.img ──▶ Kernel + Ramdisk
              │
              ▼
/init → /system/bin/init
    ▼
recovery.rc + recovery UI
```

📝 TL;DR

Yes, recovery boot **has a different flow**:

- **Loads different init scripts**
- **Uses different init binary**
- **Mounts fewer partitions**
- **Runs recovery UI**, not Android system server

Would you like a custom flowchart image summarizing this clearly across A/B vs non-A/B?

31. can a device can be A/B and non A/B?

❌ **No**, a device **cannot be both A/B and non-A/B at the same time** — these are two **mutually exclusive** partitioning and update strategies.

Let's break it down clearly:

✅ Definitions

Type	Description
A/B device	Has duplicated partitions (boot_a, system_a, vendor_a, and boot_b, system_b, etc.). Supports **seamless updates**. No separate recovery partition.
non-A/B device	Has a **single set** of partitions. Uses a **dedicated `recovery` partition** for booting into recovery. No seamless updates.

📌 Why they are mutually exclusive

Feature	A/B Device	non-A/B Device
Partitions	**_a**, **_b** suffixes	Single partitions (e.g., **`boot`**, **`system`**)
`recovery.img` partition	❌ No	✅ Yes
Seamless updates	✅ Yes	❌ No
Uses **`init_boot.img`**	✅ Yes (Android 13+)	❌ No
Fallback on update failure	✅ Yes (slot switch)	❌ Manual (recovery needed)

🔸 Android knows whether a device is A/B **at build time**, controlled by this **flag in `BoardConfig.mk`** or **`device.mk`**:

```make
AB_OTA_UPDATER := true
```

So during build:

- If **`AB_OTA_UPDATER := true`** → A/B partitions created.
- Otherwise → Non-A/B layout.

📦 Example Partition Tables

🔹 A/B Device Partition Table (e.g., Pixel)

```python-repl
boot_a
boot_b
system_a
system_b
vendor_boot_a
vendor_boot_b
...
```

🔸 non-A/B Device Partition Table

```python-repl
boot
system
vendor
recovery
...
```

🔧 Can a device be converted between them?

Not easily. It would require:

- Repartitioning (destructive)
- Rewriting bootloader support
- New OTA mechanism

So **in practice**:

- Devices are **built either A/B or non-A/B**, **never both**.

✅ Summary

> A device is **either A/B or non-A/B**. It cannot be both. The difference is fundamental to how partitions are laid out and how OTA updates work.

Let me know if you'd like a visual comparison chart of the two partition layouts.

32. means A/B is feature?

Yes ✅ — **A/B is a *feature*** (also called **"seamless update support"**) built into Android's OTA update system and bootloader logic.

📌 What kind of feature?

A/B is a **device-level and bootloader-level feature** that affects:

- How the device is **partitioned**
- How **updates are applied**
- How the system **recovers from failure**

✅ Think of A/B as a combination of:

Component	Role in A/B Feature
Partition layout	Two copies of key partitions (**`boot_a`**, **`boot_b`**, etc.)
OTA update strategy	Apply update to inactive slot, reboot into it, fallback if it fails
Bootloader support	Tracks active slot, switches on success/failure
No recovery partition	Recovery is launched from **`boot.img`** in A/B

📦 Summary:

- **A/B is a feature** designed to improve **update reliability** and avoid **bricking**.
- It requires changes in:
  - **Partitioning**
  - **OTA tools**
  - **Bootloader behavior**

✅ So yes: "A/B is a feature", but a *deep one* that impacts both software and hardware setup of a device.

Would you like a simple diagram comparing A/B vs non-A/B in visuals?

33. how to check a device support A/B

To check if an Android device supports **A/B (Seamless) Updates**, you can use any of the following methods:

✅ Method 1: Use `getprop` Command

Open a terminal (ADB shell or local terminal app) and run:

```bash
getprop ro.boot.slot_suffix
```

🔍 Output Meaning:

- If it returns `_a` or `_b` → ✅ **A/B device**
- If it returns nothing → ❌ **Non-A/B device**

✅ Method 2: Check for A/B Properties

```bash
getprop | grep slot
```

Look for:

- `ro.boot.slot_suffix`: should be `_a` or `_b`
- `ro.boot.slot`: may show `a` or `b`
- `ro.boot.dynamic_partitions`: true (usually with A/B)
- `ro.boot.dynamic_partitions_retrofit`: true (if retrofitted)

✅ Method 3: Look at Partition Table

Run:

```bash
ls -1 /dev/block/by-name/
```

🔍 A/B Device:

Will show **duplicated partitions**, like:

```nginx
boot_a
boot_b
system_a
system_b
vendor_a
vendor_b
```

🔍 Non-A/B Device:

Only single partitions:

```perl
boot
system
```

```
vendor
```

✅ Method 4: Use `ab_update` Tool (AOSP builds only)

```bash
ab_update --help
```

If the tool is available, it's an A/B-supported build.

✅ Method 5: Via Android API (Java/Kotlin)

If you're making an app:

```java
boolean isABDevice = Build.getRadioVersion().contains("_a") || Build.getRadioVersion().contains("_b");
```

Or check:

```java
String slotSuffix = SystemProperties.get("ro.boot.slot_suffix");
```

✅ Summary

Method	A/B?
`getprop ro.boot.slot_suffix`	**_a** / **_b** → ✅
Duplicate partitions (**_a**, **_b**)	✅
Recovery partition present	❌ A/B devices use **`boot.img`**
`ls -l /dev/block/by-name/`	See **_a**, **_b** partitions
`ab_update` tool	Available on A/B

Would you like a script that checks all this automatically?

34. ODM partitions

bookmark_border
Android 10 includes support for building odm partitions using the Android build system.

About ODM partitions
Original design manufacturers (ODMs) customize system-on-chip (SoC) vendor board-support packages (BSPs) to their specific devices (their boards). This enables them to implement kernel modules for board-specific components, board-specific daemons, or their own features on hardware abstraction layers (HALs). They might also want to replace or customize SoC components.

In lower Android releases, such customizations prevented the use of a single vendor image for devices with the same SoC (or with different SoCs, but in the same family). In Android 10 and higher, you can use a separate odm partition for customizations, which enables you to use a single vendor image for multiple hardware SKUs.

Use product and ODM partitions
Android 9 added support for building product partitions, enabling the use of a single system image for multiple software SKUs supplied by different product.img images. While the product partition is intended for software SKUs, the odm partition is intended for hardware SKUs.

With dedicated product and ODM partitions, you can use the system partition to host generic code for sharing among many software SKUs, and the vendor partition to host SoC-specific BSP code to share among multiple devices based on the given SoC.

Using separate partitions has disadvantages, such as the difficulty in managing disk space (for example, you must reserve a limited amount of space for future growth). However, Android 10 support for dynamic partitions removes the disk issue, and makes repartitioning a device during an over-the-air (OTA) update possible.

ODM components
The odm partition contains the following ODM-specific components (similar to the vendor partition), listed in the following table.

ODM-specific component	Location
Loadable kernel modules (LKMs)	/odm/lib/modules/*.ko
Native libraries	/odm/lib[64]
HALs	/odm/lib[64]/hw
SEPolicy	/odm/etc/selinux
VINTF object data	/odm/etc/vintf
init.rc files	/odm/etc/init
System properties	/odm/build.prop
Runtime resource overlays (RROs)	/odm/overlay/*.apk
Apps	/odm/app/*.apk
Priv-apps	/odm/priv-app/*.apk
Java libraries	/odm/framework/*.jar
Android Framework system configs	/odm/etc/sysconfig/* and /odm/etc/permissions/*

No custom images
Don't use custom images because they lack support for the following:

Installation of a module to a specific target. Custom images support copying artifacts into an image, but can't install a module into a specific partition by specifying the target partition as a part of a build rule.
Soong. custom_images can't be built using the Soong build system.
OTA update. Custom images are used as factory ROM images that can't be OTA-ed.
Maintain ABIs between partitions
The odm partition is an extension of the vendor partition. When considering application binary interface (ABI) stability, keep the following architecture in mind.

Maintaining ABI between partitions
Figure 1. Maintaining ABI between partitions.

There's no ABI stability between odm and vendor partitions. Both partitions must be upgraded at the same time.
The odm and vendor partitions can depend on each other, but the vendor partition must work without an odm partition.
The ABI between odm and system is the same as the ABI between vendor and system.
Direct interaction between the product partition and the vendor or odm partition isn't allowed. (This is enforced by SEpolicy.)

Implement ODM partitions
Before implementing a new partition, review the related AOSP changes.

Set up ODM partitions
To set up odm partitions, include these build flags:

BOARD_ODMIMAGE_PARTITION_SIZE for a fixed partition size
PRODUCT_USE_DYNAMIC_PARTITIONS and BOARD_ODMIMAGE_PARTITION_RESERVED_SIZE for a dynamic partition size
BOARD_ODMIMAGE_FILE_SYSTEM_TYPE file system type used for the ODM image
PRODUCT_ODM_PROPERTIES for /odm/build.prop for use within a $(call inherit-product path/to/device.mk), as in PRODUCT_ODM_PROPERTIES += product.abc=ok
Install a module to an ODM partition
Use these build flags to install a module to an odm partition:

device_specific: true in Android.bp
LOCAL_ODM_MODULE := true in Android.mk
Enable Verified Boot
To prevent malicious software from tampering with odm partitions, enable Android Verified Boot (AVB) for those partitions (just as you do for vendor and system partitions).

To enable AVB, include the build flag BOARD_AVB_ODM_ADD_HASHTREE_FOOTER_ARGS. For details on configuring AVB on dynamic partitions, see AVB configuration changes.

Treat /odm as another /vendor partition
To ensure that the system handles the odm partition as a vendor partition, replace any hard-coded vendor references with a set of hardware-oriented partitions (currently odm and vendor). Notable vendor reference locations in the platform include dynamic linker, package manager, and shell/libc.

Yes, you're understanding it correctly — the **`odm` partition** is like an **extension of the `vendor` partition**, intended specifically for **ODM-level customizations**. Let's break it down in **simpler words**:

🧠 What is an ODM?

- **ODM = Original Design Manufacturer**
- Think of them as companies that build the hardware based on designs provided by SoC vendors (like Qualcomm, MediaTek).

📦 What is the `odm` partition?

- A separate partition introduced in **Android 10+** to allow **hardware-specific customizations** by the ODM, without touching the **`vendor`** partition.
- This helps keep **`vendor`** **generic** and **`odm`** **device-specific**, enabling reuse of **`vendor.img`** across multiple devices using the same SoC.

🔀 Why separate it from `vendor`?

vendor partition	odm partition
Maintained by **SoC vendor** (e.g., Qualcomm)	Maintained by **ODM** (e.g., Xiaomi, Vivo)
Contains SoC-specific code	Contains board-specific and device-specific code
Shared across devices with same SoC	Unique for each device variant
Needed even without **`odm`**	**`odm`** can be optional

📁 What's stored in `/odm`?

Same kind of components as **`/vendor`**, but tailored to ODM's needs:

Component type	Location
Kernel modules (.ko)	**`/odm/lib/modules/`**
HALs	**`/odm/lib[64]/hw/`**
Native libraries	**`/odm/lib/`**
init scripts	**`/odm/etc/init/*.rc`**
SEPolicy configs	**`/odm/etc/selinux/`**
VINTF info	**`/odm/etc/vintf/`**
System properties	**`/odm/build.prop`**

Component type	Location
Apps / Overlays	`/odm/app/`, `/odm/overlay/`

🧩 ABI Compatibility Notes:

- `odm` and `vendor` can **depend on each other**, but must be **updated together**.
- No ABI stability between `odm` and `vendor` (you can break APIs between them).
- ABI between `odm` and `system` **must be stable**, like `vendor ↔ system`.

🔐 Verified Boot Protection

You can protect the `odm` partition with **AVB (Android Verified Boot)**:

```makefile
BOARD_AVB_ODM_ADD_HASHTREE_FOOTER_ARGS := ...
```

🛠️ Build Support

To use `odm` in your device configuration:

```makefile
# For fixed partition
BOARD_ODMIMAGE_PARTITION_SIZE := <size>

# For dynamic partition
PRODUCT_USE_DYNAMIC_PARTITIONS := true
BOARD_ODMIMAGE_PARTITION_RESERVED_SIZE := <size>

# Filesystem type
BOARD_ODMIMAGE_FILE_SYSTEM_TYPE := ext4
```

To install something to `/odm`:

- In Android.mk:

```makefile
LOCAL_ODM_MODULE := true
```

- In Android.bp:

```bp
device_specific: true
```

💡 Summary

- `odm` is for device-specific **hardware customizations**
- Keeps `vendor` clean and reusable across devices
- Introduced in Android 10
- Works with dynamic partitions
- Must be verified with AVB for security
- Optional, but helpful in modularizing device software

Would you like me to generate a table comparing `/system`, `/vendor`, `/odm`, and `/product` too?

35. Product partitions

bookmark_border
Android 9 and higher includes support for building product partitions using the Android build system. Previously, Android 8.x enforced the separation of SoC-specific components from the system partition to the vendor partition without dedicating space for OEM-specific components built from the Android build system. Android 9 and higher provides additional permissions and whitelisting features that apply to priv-apps on different partitions.

About product partitions
Many OEMs customize the AOSP system image to implement their own features, as well as carrier requirements. However, such customizations make it impossible to use a single system image for multiple software SKUs. Each image must be different to support the customizations, such as with different locales or carriers. Using a separate product partition to contain customizations makes it possible to use a single system image for multiple software SKUs. (The system partition hosts generic code that can be shared among many software SKUs). The vendor partition continues to host SoC-specific BSP code which can be shared among multiple devices based on the given SoC.

Using separate partitions has some disadvantages, such as managing disk space (a limited amount of space must remain reserved for future growth) and maintaining a stable application binary interface (ABI) between partitions. Before deciding to use product partitions, take time to consider your unique AOSP implementation and possible mitigation tactics (such as repartitioning a device during an over-the-air (OTA) update, which isn't done by Google but is done by some OEMs). Dynamic partitioning is a good solution for this.

Product partitions and permissions
In Android 9 and higher, a change in the permissions and whitelisting process affects how you grant priv-apps permissions on your `product` partitions. The permissions.xml file must reside in the same partition as the priv-apps. Placing a permissions.xml file in the system partition for priv-apps doesn't extend those permissions to priv-apps in the product partition, even though the former is an extension of the latter. For details on the permissions and whitelisting processes, see Privileged Permission Allowlisting.

Legacy /oem versus /product
We have two kinds of attributes of the product partition depending on the product interface enforcement. Also, the product partition is different than the legacy oem partition:

Partition Attributes
oem
Not updateable; usually flashed once at the factory.
Built per small variations, such as branding and color. Having different oem partition contents doesn't mean the product software is different.
The system partition doesn't depend on the oem partition. (It uses the oem partition only when a specific file is found there).
Only uses public API on the system partition.
product
Updateable
Coupled with the system image (they update together)
Built per product or product families.
System partition can depend on product partition.
Can use non-public APIs since they are updated simultaneously.
product (enforced interfaces)
Updateable
Decoupled with the system image.
Built per product or product families.
The system partition doesn't depend on the product partition.
Can't use hidden APIs, but only uses public and system APIs on the system partition.
For these reasons, Android 9 supports the product partition while retaining support for the legacy oem partition, for devices that depend on it. To decouple the product partition from the system partition, Android 11 supports enforcing product interfaces.

/product components
The product partition contains the following components:

Product-specific system properties (/product/build.prop)
Product-specific RROs (/product/overlay/*.apk)
Product-specific apps (/product/app/*.apk)
Product-specific priv-apps (/product/priv-app/*.apk)
Product-specific libraries (/product/lib/*)
Product-specific java libraries (/product/framework/*.jar)
Product-specific Android Framework system configs (/product/etc/sysconfig/* and /product/etc/permissions/*)
Product-specific media files (/product/media/audio/*)
Product-specific bootanimation files

No custom_images
You can't use custom_images. They lack support for the following:

Installing modules into a specific target. custom_images support copying artifacts into an image but can't install a module into a specific partition by specifying its target partition as a part of a build rule.
Soong support. custom_images can't be built using the Soong build system.
OTA update support. custom_images are used as factory ROM images that can't receive OTA updates.
Maintain ABIs between partitions
The product partition in Android 9 is an extension of the system partition. There's a weak ABI between product and system partitions, so both must be upgraded at the same time, and the ABI should be system SDK-based. If the system SDK doesn't cover all API surfaces between product and system, OEMs must maintain their own ABIs between the two partitions.

The product and system partitions can have dependency on each other. However, tests with the Generic System Image (GSI) must work properly without the product partition.

When the product interfaces are enforced, the product partition is decoupled with the system partition. The product partition uses only the allowed interfaces from the system partition.

The product partition must not have any dependency through unstable interfaces on vendor partition. Direct interaction between the product and vendor partitions is forbidden. (This is enforced by SEpolicy.)

Implement product partitions
Before implementing a new product partition, review the related product partition changes in AOSP. Then, to set up product, include the following board or product-build flags:

BOARD_USES_PRODUCTIMAGE
BOARD_PRODUCTIMAGE_PARTITION_SIZE
BOARD_PRODUCTIMAGE_FILE_SYSTEM_TYPE
PRODUCT_PRODUCT_PROPERTIES for /product/build.prop. These must be within a $(call inherit-product path/to/device.mk), as in PRODUCT_PRODUCT_PROPERTIES += product.abc=ok.
Install a module to the product partition
Use the following build flags to install a module to the product partition.

product_specific: true in Android.bp
LOCAL_PRODUCT_MODULE := true in Android.mk
Enable Verified Boot
To prevent the product partition from being tampered with by malicious software, enable Android Verified Boot (AVB) for that partition (just as you do for the vendor and system partitions). To enable AVB, include the following build flags:
BOARD_AVB_PRODUCT_ADD_HASHTREE_FOOTER_ARGS.

Great! Let's simplify and summarize what **`product` partitions** are in Android (especially Android 9+), how they work, and how they differ from `system`, `vendor`, `odm`, and `oem`.

✅ What is the `product` partition?

It's a **separate partition** introduced in **Android 9+** to hold **OEM-specific software customizations**, such as:

- Features for different carriers (e.g., Verizon, AT&T)
- Locale-based differences (e.g., language-specific apps)
- Custom apps, RROs, libraries, audio, etc.
  - 🔸 This helps share a single `system.img` across multiple **software SKUs**.

💡 Why `product` is needed?

Before Android 9:

- All customizations went into `system`, making it device-specific.

After Android 9:

- `system` remains **generic**

- `product` stores **OEM-specific and carrier-specific** customizations

This modular design supports:

- Shared `system.img` across product variants
- Better OTA management
- Easier reuse and maintenance

📦 What goes into `/product`?

Component Type	Path
System properties	`/product/build.prop`
Runtime Resource Overlays (RROs)	`/product/overlay/*.apk`
Apps	`/product/app/*.apk`
Privileged apps	`/product/priv-app/*.apk`
Native libraries	`/product/lib/*`
Java framework libs	`/product/framework/*.jar`
System configs (sysconfig, perms)	`/product/etc/sysconfig/*`, `/permissions/*`
Audio/media/bootanimation	`/product/media/audio/`, etc.

🆚 Difference: `product` vs `oem` (legacy)

Feature	`/product`	`/oem` (legacy)
Updatable via OTA	✅ Yes	❌ No (factory-flashed only)
Coupled with `system`	✅ Usually	❌ Independent
Build target	Per product or family	Per SKU (branding, color)
Hidden API usage	✅ Yes (unless interface is enforced)	❌ Only public APIs
Partition type	Part of AOSP	Often factory only

> Android 11 supports **interface-enforced product partitions**, which **decouple** `product` and `system`.

🔐 AVB Support

To protect `product`, enable **AVB (Android Verified Boot)**:

```makefile
BOARD_AVB_PRODUCT_ADD_HASHTREE_FOOTER_ARGS := ...
```

⚙️ How to enable and configure `product` partition?

In `BoardConfig.mk`:

```makefile
BOARD_USES_PRODUCTIMAGE := true
BOARD_PRODUCTIMAGE_PARTITION_SIZE := <size>
BOARD_PRODUCTIMAGE_FILE_SYSTEM_TYPE := ext4
```

In `device.mk` or `product.mk`:

```makefile
PRODUCT_PRODUCT_PROPERTIES += product.feature.enabled=true
```

📥 Install apps/modules to `/product`

In `Android.mk`:

```
makefile

LOCAL_PRODUCT_MODULE := true
```

In `Android.bp`:

```
bp

product_specific: true
```

🔄 ABI Rules (Product ↔ System)

Rule	Description
✅ Can depend on **`system`**	But must upgrade together unless interface-enforced
⚠️ Weak ABI stability	Maintain SDK-based ABI or define your own
🚫 Cannot access **`vendor`** directly	Enforced by SEPolicy
✅ Must work without **`/product`** for GSI testing	System image must boot standalone

---

🔧 Interface Enforcement (Android 11+)

If you want `/product` to be **independent** of `/system`, enable **product interface enforcement**.

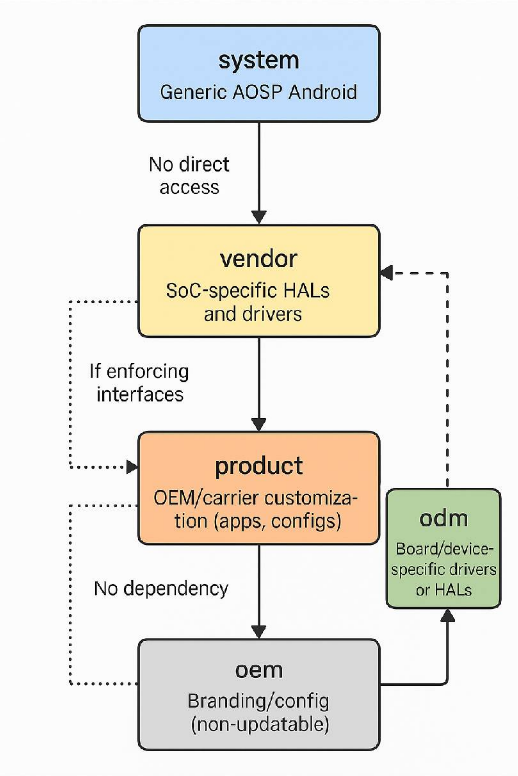
Interface type	System → Product	Product → System
Public	✅ Allowed	✅ Allowed
Hidden	❌ Not allowed	❌ Not allowed

🧠 Summary Table

Partition	Purpose	Maintained by	OTA Updateable	Shared across devices?
`system`	Generic AOSP Android	Google / OEM	✅	✅
`vendor`	SoC-specific HALs and drivers	SoC vendor	✅	✅
`odm`	Board/device-specific drivers or HALs	ODM	✅	❌
`product`	OEM/carrier customization (apps, configs)	OEM	✅	❌ (per software SKU)
`oem`	Branding/config (non-updatable)	Factory only	❌	✅

Would you like a **diagram showing how these partitions interact** (like arrows for dependencies)?

36. YES


system
Generic AOSP Android
No direct access
vendor
SoC-specific HALs and drivers
If enforcing interfaces
product
OEM/carrier customiza- tion (apps, configs)
odm
Board/device-specific drivers or HALs
No dependency
oem
Branding/config (non-updatable)

37. what is the use of interfaces between these partitions and why it exist

Interfaces between partitions (like **system** ↔ **vendor**, **system** ↔ **product**, or **system** ↔ **odm**) exist to:

✅ Purpose of Interfaces Between Partitions

1. **Enable Modularity and Separation of Concerns**
    - Each partition serves a different purpose:
        - `system`: AOSP generic framework.
        - `vendor`: SoC-specific HALs and binaries.
        - `odm`: Board/device-specific customizations.
        - `product`: OEM/carrier/product-specific features.
    - Interfaces allow these to work together **without merging code**.
2. **Allow Independent Development and Updates**
    - SoC vendors, OEMs, and Google work **independently** on their partitions.
    - Interfaces define **what APIs or behavior each side can depend on**.
    - Example: A `vendor` module should work even if `system` is updated.
3. **Ensure Compatibility Across Versions (VTS/CTS/GSI)**
    - Interfaces (like **HIDL**, **AIDL**, or **SDK/NDK APIs**) allow testing and validation:
        - Vendor Test Suite (VTS) for vendor partition
        - Compatibility Test Suite (CTS) for system
        - Generic System Image (GSI) for enforcing standard interface
4. **Support for OTA Updates (e.g., Seamless A/B)**
    - Since partitions can be updated **separately**, interfaces ensure:
        - `system.img` v1 can work with `vendor.img` v0
        - Future-proofing across Android versions
5. **Reduce Fragmentation**
    - Google enforces **stable, versioned interfaces** (like VINTF, HAL manifests) so devices can stay compatible with newer Android versions without breaking.

🧠 Why Not Just Use One Partition?

Because different vendors:

- Work on different timelines
- Use proprietary or licensed code
- May not want to share or expose internal changes

Interfaces allow collaboration **without breaking encapsulation**.

📌 Summary Table

Partition Pair	Interface Purpose	Example
`system` ↔ `vendor`	HALs (HIDL/AIDL), VINTF	Camera HAL, sensors
`system` ↔ `product`	SDK APIs (or enforced product interfaces)	OEM settings app
`system` ↔ `odm`	Same as `vendor` (device-specific HALs)	Touchscreen driver
`vendor` ↔ `odm`	Tight coupling allowed	Same SoC family
`product` ↔ `vendor`	❌ Not allowed (blocked by SEpolicy)	—

Let me know if you want a visual version of this explanation.

38. Implement a GKI module partition

bookmark_border
GKI and GKI modules can be updated independently from the rest of the partition because GKI modules reside on a separate dynamic partition in the super image called system_dlkm. GKI modules are signed by Google using the kernel build-time key pair and are compatible only with the GKI they're built with. There's no ABI stability between GKI and GKI modules; for modules to load correctly during runtime, GKI and GKI modules must be built and updated together.

Note: Any device launching with Android 13 and higher must include a system_dlkm partition.
Implement system_dklm partition support
The system_dlkm partition is located in the super partition as another dynamic partition. This partition can contain:

Google build-time signed kernel modules
depmod artifacts
Build system_dlkm
Building system_dlkm is a similar process to building other dynamic partitions. Perform the following steps to add system_dlkm to your build:

In BoardConfig.mk, add the following entries:

```
BOARD_USES_SYSTEM_DLKMIMAGE := true
BOARD_SYSTEM_DLKMIMAGE_FILE_SYSTEM_TYPE := $(TARGET_RO_FILE_SYSTEM_TYPE)
TARGET_COPY_OUT_SYSTEM_DLKM := system_dlkm
```
In the partition list, add system_dlkm: BOARD_GOOGLE_SYSTEM_DYNAMIC_PARTITIONS_PARTITION_LIST := system_dlkm

(Optional) For A/B and virtual A/B devices, add the following line in the device.mk file for your device:

```
AB_OTA_PARTITIONS += system_dlkm
```
Identify kernel modules to copy into system_dlkm
For modules to load successfully at runtime, GKI and GKI modules must be built together. Therefore you must identify kernel modules in the GKI build for the target architecture and provide that as source for the system_dlkm partition during platform build.

For Android 13
Point BOARD_SYSTEM_DLKM_SRC to a folder containing the required GKI modules kernel object files for the device as an input to the build system to generate the system_dlkm partition. For example:

Provide the GKI modules source in a folder and point BOARD_SYSTEM_DLKM_SRC to that folder. For example:

```
  BOARD_SYSTEM_DLKM_SRC := kernel/prebuilts/5.10/arm64/system_dlkm_staging
```
At build time, modules listed in BOARD_SYSTEM_DLKM_SRC are installed in $ANDROID_PRODUCT_OUT/system_dlkm.

For Android 14
We have streamlined the implementation with the macros (BOARD_*_KERNEL_MODULES) being used for other *_dlkm partitions. The list of required GKI modules for the device should be referenced by BOARD_SYSTEM_KERNEL_MODULES macro. At build time these modules are installed in the $ANDROID_PRODUCT_OUT/system_dlkm. Any module in vendor_dlkm partition which has dependencies on the modules in system_dlkm partition generates correct references in modules.dep file for the vendor_dlkm partition. Due to the cross-partition dependencies represented by modules.dep, when a vendor module gets loaded, any required GKI module is loaded automatically.

For example, to install all GKI modules on system_dlkm partition for GKI arm64 kernel 5.15 from prebuilts:

```
 BOARD_SYSTEM_KERNEL_MODULES := $(wildcard kernel/prebuilts/5.15/arm64/*.ko)
```
Mount system_dlkm at runtime
Depending on the file system being used as a read-only file system, add the following in your fstab to mount the system_dlkm partition at runtime:

ext4 as a read-only file system

```
  system_dlkm /system_dlkm ext4 noatime,ro,errors=panic wait,logical,first_stage_mount,slotselect,avb
```
erofs as read-only file system

system_dlkm /system_dlkm erofs ro wait,logical,first_stage_mount,slotselect,avb
Partition mounting and module loading
During first_stage_init, the system_dlkm partition is mounted in the /system_dlkm as a read-only file system. On a successful mount, symbolic links at /system/lib/modules pointing to /system_dlkm/lib/modules are available.

A vendor process, such as an .rc script, can then load the kernel modules based on the order specified in modules.load. The vendor process must use the symbolic link /system/lib/modules to load the modules. If necessary, the vendor process can also load the modules at a later time.

SELinux
Every file in the system_dlkm partition is labeled with the file context of system_dlkm_file. To load the GKI modules file in the system_dlkm partition, the vendor process responsible for loading the modules needs a sepolicy in the vendor domain.

For example, dlkm_loader used by Cuttlefish to load GKI modules has the following permissions in the policy file at shared/sepolicy/vendor/dlkm_loader.te:

allow dlkm_loader self:capability sys_module;
allow dlkm_loader system_dlkm_file:dir r_dir_perms;
allow dlkm_loader system_dlkm_file:file r_file_perms;
allow dlkm_loader system_dlkm_file:system module_load;
Validate the system-dlkm partition
Google provides a GKI VTS test case to verify the system_dlkm partition. To manually invoke the test, use the following atest command:

atest -c vts_dlkm_partition_test

Great — you're diving deep into **GKI and the `system_dlkm` partition**. Here's a clear breakdown of:

✅ Why Interfaces Exist Between GKI (kernel image) and system_dlkm (kernel modules)

🔧 Reason 1: Modularity and Maintainability

- **`system_dlkm`** is a **dynamic partition** that contains GKI kernel modules.
- These modules are **not built into the `boot` image** (which contains the core GKI kernel), but **stored separately**, making them modular.
- It allows:
  - Independent updates to kernel modules
  - Smaller boot images
  - OEM flexibility

🧩 Reason 2: Defined Interfaces Allow Separation

- There is **no ABI (binary interface) stability between GKI and modules**.
  → So, modules **must match exactly** the GKI version they are compiled against.
- Because of this, the interface is **not stable**, but it **must exist** so the kernel can:
  - Dynamically load **`.ko`** modules
  - Resolve symbols correctly via **`modules.dep`**
  - Maintain order via **`modules.load`**

📦 Reason 3: OTA and A/B Update Compatibility

- Since **`system_dlkm`** is a dynamic partition, it can be updated in OTA (especially A/B devices).
- Interfaces (via **`.dep`** and **`.load`**) make sure:
  - Kernel modules load in proper order
  - Dependencies across **`system_dlkm`** and **`vendor_dlkm`** are respected

🔒 Reason 4: Security, Isolation & Verified Boot

- `system_dlkm` uses **AVB (Android Verified Boot)** to ensure integrity.
- SEPolicy enforces:
  - Which processes can load from `system_dlkm`
  - What capabilities are required (`sys_module`, `read`, `execute`)
- Example: `dlkm_loader` can load modules if policy allows it.

📚 Summary: Why Interfaces Exist Between Partitions Like GKI ↔ system_dlkm

Feature	Purpose
Module location (`/system_dlkm`)	Stores Google-signed `.ko` modules outside of the boot image
Interface type	Symbol resolution (`modules.dep`), loading order (`modules.load`)
Dynamic linking	Kernel uses `insmod`/`modprobe` to load modules with symbol references
ABI stability	No — GKI and modules must be built together
OTA-friendly	Yes — Dynamic partition allows seamless OTA of modules
SEPolicy control	Enforces who can load modules from system_dlkm
Verified Boot	AVB hash tree validates system_dlkm

Would you like a **visual diagram** of how GKI and system_dlkm relate and interact during boot and runtime?

39. ↪
`system_dlkm`
why it is required and when introduced

Great question. Let's dive into **what `system_dlkm` is**, **why it's required**, and **when it was introduced**.

🧠 What is `system_dlkm`?

`system_dlkm` is a **dynamic partition** introduced to store **loadable kernel modules (LKMs)** that are part of the **GKI (Generic Kernel Image)**.

- `dlkm` = **Dynamic Loadable Kernel Modules**
- These `.ko` files are signed and separated from the monolithic boot image
- `system_dlkm` → used **only** for GKI-signed modules (not vendor ones)

🎯 Why is `system_dlkm` Required?

1. GKI Modularity

- Android introduced the **Generic Kernel Image (GKI)** to separate core kernel from device-specific customization.
- GKI reduces fragmentation and improves updatability across Android devices.

But kernel modules are still needed. Instead of baking them into the boot image, Google moved them to `system_dlkm`.

✅ Benefit: Kernel updates become easier and smaller.

---

2. OTA Updates

- `system_dlkm` is a **dynamic partition** → can be **OTA-updated independently** of `boot`.
- This means you can update your `.ko` files (e.g. GPU, storage driver) without touching `boot`.

✅ Benefit: Safe and seamless kernel module updates, especially for **A/B partitioned** devices.

---

3. Security & Verified Boot

- Files in `system_dlkm` are **AVB-verified**, and SELinux tightly controls access.
- Prevents unsigned modules or tampering.

✅ Benefit: Secures module loading and boot integrity.

---

4. Cross-Partition Dependencies

- With new partitioning like `vendor_dlkm`, modules may depend on each other.
- Using `modules.dep`, the kernel resolves dependencies even across `system_dlkm` ↔ `vendor_dlkm`.

✅ Benefit: Keeps clean layering between Google's GKI and vendor components.

📅 When Was `system_dlkm` Introduced?

Android Version	Status of `system_dlkm`
Android 11	Optional, early support
Android 12	More widely adopted
Android 13	**Mandatory** for new devices
Android 14+	Streamlined integration using macros

> 📌 **Official Requirement**: Any device launching with **Android 13 or later must** include a `system_dlkm` partition.

🔄 Summary

Feature	Description
Purpose	Holds Google-signed kernel modules separately
Partition type	Dynamic partition (in super partition)
Introduced	Android 11 (optional), **mandatory from Android 13**
Benefits	OTA modularity, GKI compliance, AVB verification
Required for GKI?	✅ Yes – for separation of generic and vendor modules
Mount location	**`/system_dlkm`** → symlinked to **`/system/lib/modules`**

Would you like an annotated **boot-time flowchart showing GKI, system_dlkm, and vendor_dlkm interactions**?

40. Enforce product partition interfaces

bookmark_border
Android 11 unbundles the product partition, making it independent of the system and vendor partitions. As part of these changes, you can now control the product partition's access to native and Java interfaces (which is similar to how interface enforcement works for vendor partitions).

Enforce native interfaces
To enable the native interface enforcement, set PRODUCT_PRODUCT_VNDK_VERSION to current. (The version is automatically set to current when the shipping API level for the target is greater than 29.) Enforcement allows:

Native modules in the product partition to link:
Statically or dynamically to other modules in the product partition that include static, shared, or header libraries.
Dynamically to VNDK libraries in the system partition.
JNI libraries in unbundled APKs in the product partition to link to libraries in /product/lib or /product/lib64 (this is in addition to the NDK libraries).
Enforcement doesn't allow other links to partitions other than the product partition.

Build time enforcement (Android.bp)
In Android 11, system modules can create a product image variant in addition to core and vendor image variants. When native interface enforcement is enabled (PRODUCT_PRODUCT_VNDK_VERSION is set to current):

Native modules in the product partition are in the product variant instead of the core variant.

Modules with product_available: true in their Android.bp files are available to the product variant.

Libraries or binaries that specify product_specific: true can link to other libraries that specify product_specific: true or product_available: true in their Android.bp files.

VNDK libraries must have product_available: true in their Android.bp files so product binaries can link to VNDK libs.

The following table summarizes the Android.bp properties used to create image variants.

Properties in Android.bp Variants created
Before enforcement After enforcement
default (none) core
(includes /system, /system_ext and /product) core
(includes /system and /system_ext but not /product)
system_ext_specific: true core core
product_specific: true core product
vendor: true vendor vendor
vendor_available: true core, vendor core, vendor
product_available: true N/A core, product
vendor_available: true AND product_available: true N/A core, product, vendor
system_ext_specific: true AND vendor_available: true core, vendor core, vendor
product_specific: true AND vendor_available: true core, vendor product, vendor
Note: The hidl_interface implies both product_available: true and vendor_available: true without explicitly including it in Android.bp, making hidl_interface modules with or without system_ext_specific: true available to all partitions.
Build time enforcement (Android.mk)
When native interface enforcement is enabled, native modules installed to the product partition have a native:product link type that can link only to other native:product or native:vndk modules. Attempting to link to any modules other than these causes the build system to generate a link type check error.

Runtime enforcement
When native interface enforcement is enabled, the linker configuration for the bionic linker doesn't allow system processes to use product libraries, creating a product section for the product processes that can't link to libraries outside the product partition (however, such processes can link to VNDK libraries). Attempts to violate the runtime link configuration cause the process to fail and generate a CANNOT LINK EXECUTABLE error message.

Enforce Java interfaces
To enable the Java interface enforcement, set PRODUCT_ENFORCE_PRODUCT_PARTITION_INTERFACE to true. (The value is automatically set to true when the shipping API level for the target is greater than 29.) When enabled, enforcement allows or disallows the following access:

API /system /system_ext /product /vendor /data
Public API

@SystemApi
@hide API
As in the vendor partition, an app or a Java library in the product partition is allowed to use only public and system APIs; linking to a library that uses hidden APIs isn't allowed. This restriction includes linking at build time and reflection in runtime.

Build time enforcement
At build time, Make and Soong verify that Java modules in the product partition don't use hidden APIs by checking the platform_apis and sdk_version fields. The sdk_version of apps in the product partition must be filled with current, system_current, or numeric version of the API, and the platform_apis field must be empty.

Runtime enforcement
The Android runtime verifies that apps in the product partition don't use hidden APIs, including reflection. For details, refer to Restrictions on non-SDK interfaces.

Note: Use the veridex static analysis tool to scan apps on a device and locate hidden API usages. For details, refer to Test using the veridex tool.
Enable product interface enforcement
Use the steps in this section to enable product interface enforcement.

Step	Task	Required
1	Define your own system makefile that specifies the packages for the system partition, then set the artifacts path requirement check in the device.mk (to prevent nonsystem modules from installing to the system partition).	N
2	Clean up the allowed list.	N
3	Enforce native interfaces and identify runtime link failures (can run in parallel with Java enforcement).	Y
4	Enforce Java interfaces and verify runtime behavior (can run in parallel with native enforcement).	Y
5	Check runtime behaviors.	Y
6	Update device.mk with product interface enforcement.	Y

Step 1: Create makefile and enable artifact path check
In this step, you define the system makefile.

Create a makefile that defines the packages for the system partition. For example, create an oem_system.mk file with the following:

```
$(call inherit-product, $(SRC_TARGET_DIR)/product/handheld_system.mk)
$(call inherit-product, $(SRC_TARGET_DIR)/product/telephony_system.mk)

# Applications
PRODUCT_PACKAGES += \
  CommonSystemApp1 \
  CommonSystemApp2 \
  CommonSystemApp3 \

# Binaries
PRODUCT_PACKAGES += \
  CommonSystemBin1 \
  CommonSystemBin2 \
  CommonSystemBin3 \

# Libraries
PRODUCT_PACKAGES += \
  CommonSystemLib1 \
  CommonSystemLib2 \
  CommonSystemLib3 \

PRODUCT_SYSTEM_NAME := oem_system
PRODUCT_SYSTEM_BRAND := Android
PRODUCT_SYSTEM_MANUFACTURER := Android
PRODUCT_SYSTEM_MODEL := oem_system
PRODUCT_SYSTEM_DEVICE := generic

# For system-as-root devices, system.img should be mounted at /, so we
# include ROOT here.
_my_paths := \
 $(TARGET_COPY_OUT_ROOT)/ \
 $(TARGET_COPY_OUT_SYSTEM)/ \

$(call require-artifacts-in-path, $(_my_paths),)
```

In the device.mk file, inherit the common makefile for the system partition and enable the artifact path requirements check. For example:

```
$(call inherit-product, $(SRC_TARGET_DIR)/product/oem_system.mk)

# Enable artifact path requirements checking
PRODUCT_ENFORCE_ARTIFACT_PATH_REQUIREMENTS := strict
```

About the artifact path requirements
When PRODUCT_ENFORCE_ARTIFACT_PATH_REQUIREMENTS is set to true or strict, the build system prevents packages defined in other makefiles from installing to the paths defined in require-artifacts-in-path and prevents packages defined in the current makefile from installing artifacts outside the paths defined in require-artifacts-in-path.

In the example above, with PRODUCT_ENFORCE_ARTIFACT_PATH_REQUIREMENTS set to strict, makefiles outside oem_system.mk can't include modules installed to the root or system partition. To include these modules, you must either define them in the oem_system.mk file itself or in an included makefile. Attempts to install modules to disallowed paths cause build breaks. To fix breaks, do one of the following:

Option 1: Include the system module in the makefiles included in oem_system.mk. This makes it so the artifact path requirement is met (as the modules now exist in an included makefile) and thus allows installation to the set of paths in **`require-artifacts-in-path`**.

Option 2: Install modules to the system_ext or product partition (and don't install modules to the system partition).

Option 3: Add modules to the PRODUCT_ARTIFACT_PATH_REQUIREMENT_ALLOWED_LIST. This lists allowed modules to be installed.

Step 2: Empty the allowed list
In this step, you make the PRODUCT_ARTIFACT_PATH_REQUIREMENT_ALLOWED_LIST empty so all devices sharing oem_system.mk can also share a single system image. To empty the allowed list, move any modules in the list to the system_ext or product partition or add them to system make files. This step is optional because defining a common system image isn't required to enable product interface enforcement. However, emptying the allowed list is helpful for defining the system boundary with system_ext.

Step 3: Enforce native interfaces
In this step, you set PRODUCT_PRODUCT_VNDK_VERSION := current, then look for build and runtime errors and resolve them. To check the device boot and logs and find and fix runtime link failures:

Set PRODUCT_PRODUCT_VNDK_VERSION := current.

Build the device and look for build errors. You're likely to see a few build breaks for missing product variants or core variants. Common breaks include:

Any hidl_interface module that has product_specific: true won't be available for system modules. To fix, replace product_specific: true with system_ext_specific: true.
Modules might be missing the product variant required for product modules. To fix, make that module available to the product partition by setting product_available: true or move the module to the product partition by setting product_specific: true.
Note: Build breaks are similar for modules described in Android.mk.
Resolve build errors and ensure that the device builds successfully.

Flash the image and look for runtime errors in the device boot and logs.

If the linker tag from a test case log shows a CANNOT LINK EXECUTABLE message, the make file is missing a dependency (and wasn't captured at build time).
To check it from the build system, add the required library to the shared_libs: or required: field.
Resolve the missing dependencies using the guidance given above.

Step 4: Enforce Java interfaces
In this step, you set PRODUCT_ENFORCE_PRODUCT_PARTITION_INTERFACE := true, then find and fix resultant build errors. Look for two specific types of errors:

Link type errors. This error indicates that an app links to Java modules that have a broader sdk_version. To fix, you can broaden the app's sdk_version or restrict the library's sdk_version. Example error:

```
error: frameworks/base/packages/SystemUI/Android.bp:138:1: module "SystemUI" variant "android_common": compiles against system API, but dependency "telephony-common" is compiling against private API.Adjust sdk_version: property of the source or target module so that target module is built with the same or smaller API set than the source.
```

Symbol errors. This error indicates that a symbol can't be found because it's in a hidden API. To fix, use a visible (non-hidden) API or find an alternative. Example error:

```
frameworks/opt/net/voip/src/java/com/android/server/sip/SipSessionGroup.java:1051: error: cannot find symbol
            ProxyAuthenticate proxyAuth = (ProxyAuthenticate)response.getHeader(
                                          ^
  symbol:   class ProxyAuthenticate
  location: class SipSessionGroup.SipSessionImpl
```

Step 5: Check runtime behaviors
In this step, you verify runtime behaviors are as expected. For apps that are debuggable, you can monitor hidden API usage by log using StrictMode.detectNonSdkApiUsage (which generates a log when the app uses a hidden API). Alternatively, you can use the veridex static analysis tool to get the type of usage (linking or reflection), restriction level, and call stack.

Veridex syntax:

```
./art/tools/veridex/appcompat.sh --dex-file={apk file}
```

Example veridex result:

```
#1: Linking greylist-max-o Landroid/animation/AnimationHandler;-><init>()V use(s):
       Lcom/android/systemui/pip/phone/PipMotionHelper;-><init>
(Landroid/content/Context;Landroid/app/IActivityManager;Landroid/app/IActivityTaskManager;Lcom/android/systemui/pip/phone/PipMenuActivityController;Lcom/android/internal/policy/PipSnapAlgorithm;Lcom/android/systemui/statusbar/FlingAnimationUtils;)V

#1332: Reflection greylist Landroid/app/Activity;->mMainThread use(s):
       Landroidx/core/app/ActivityRecreator;->getMainThreadField()Ljava/lang/reflect/Field;
```

For details on veridex usage, refer to Test using the veridex tool.

Step 6: Update device.mk
After fixing all build and runtime failures, and verifying that runtime behaviors are as expected, set the following in device.mk:

```
PRODUCT_PRODUCT_VNDK_VERSION := current
PRODUCT_ENFORCE_PRODUCT_PARTITION_INTERFACE := true
```

Great! Let's now **explain in simple terms**:

🔒 What Does "Enforcing Product Partition Interfaces" Mean?

When Android enforces interfaces for the **`/product` partition**, it **restricts what it can talk to**, just like **sandboxing**. It means:

> 📦 Code in `/product` must **only access allowed APIs and libraries**, not just grab anything from `/system`, `/vendor`, or use hidden internals.

✅ Why Is This Interface Enforcement Important?

1. Modularity

Android is trying to **separate system logic** from **device/product-specific logic**. By keeping `/product` separate:

- System partition stays **generic and reusable**
- Product partition can have OEM/brand customizations
- Easier OTA updates and long-term maintenance

---

2. Stability and Compatibility

By enforcing which APIs or libraries `/product` code can access:

- System updates won't break `/product` apps (and vice versa)
- Ensures **ABI** (binary interface) compatibility
- Helps with **GSI testing** (Generic System Image can boot without product code)

---

3. Security

- Prevents `/product` code from using **private, hidden APIs** in `/system`
- Protects against fragile or unintended behaviors

🧱 What Are These "Interfaces"?

They are the **"rules of communication"** between partitions:

Interface Type	Description
Native Interface	Shared object (.so) libraries, C/C++ code
Java Interface	Java SDKs, hidden APIs, reflection
Runtime Linker	What native code can link to at runtime
SDK Enforcements	Which Java APIs are visible to apps at build and runtime

---

📅 When Did This Start?

Android Version	What Happened
Android 9	Introduced `/product` partition (optional)
Android 11	**Enforced interfaces** between `/product` ↔ others
Android 13+	Stronger build/runtime checks & required interface enforcement

---

⚙️ How Is It Enforced?

1. Native Interface Enforcement

- Done by setting:

```
make

PRODUCT_PRODUCT_VNDK_VERSION := current
```

- Product binaries can only link to:
  - Other `/product` libs
  - VNDK libraries in `/system/lib/vndk*`

❌ Can't link to arbitrary `/system` or `/vendor` libs

---

2. Java Interface Enforcement

- Done by setting:

```
make

PRODUCT_ENFORCE_PRODUCT_PARTITION_INTERFACE := true
```

- Apps/libraries in `/product`:
  - Can only use **public** and **@SystemApi**
  - ❌ Cannot use **@hide** or private APIs
  - ❌ No reflection into private APIs

🔍 How Are Violations Caught?

Phase	Tools/Methods
Build time	Soong/Make system shows linker or symbol errors
Runtime	Crashes with `CANNOT LINK EXECUTABLE`, or logs API misuse
Static scan	Use `veridex` tool to find hidden API usage in apps

🚧 Example Errors

🔗 Linker Error:

```csharp
CANNOT LINK EXECUTABLE: library "libx.so" needed, but inaccessible from product
```

🔐 Java Symbol Error:

```arduino
cannot find symbol class ProxyAuthenticate
```

📜 Veridex Output:

```bash
Reflection greylist Landroid/app/Activity;->mMainThread
```

🛠️ How to Fix Violations?

- **Use only public or system APIs**
- **Add `product_available: true`** to required native libraries
- **Don't use `@hide` APIs or private symbols**
- Refactor code to move parts into `/system_ext` if needed

✅ Summary Table

Feature	Enforced?	Purpose
Native linking	✅	`/product` can only use certain libraries
Java hidden API access	✅	Apps can't use non-public Java APIs
Cross-partition protection	✅	Enforced by linker and SELinux
Build-time enforcement	✅	Errors if rules violated
Runtime enforcement	✅	Crashes or blocked loading
Tools	`veridex`, `StrictMode`, logs	

Let me know if you want a **visual flowchart of these rules**, or a **step-by-step checklist** to apply enforcement on your own custom device.

41. is this interface enforcement exist both way ?

Excellent question! Let's break this down **clearly**:

🔄 Does Interface Enforcement Work Both Ways?

❌ No. Interface enforcement is NOT fully bidirectional.

It is **one-directional**, meaning **who can access what** is **strictly defined** per partition.

🔍 Let's Understand the Access Rules

From → To	Is Access Allowed?	Notes
`/product` → **`/system`** (core)	✅ **Limited** access	Only public & VNDK libs, or via product-available
`/system` → **`/product`**	❌ **No access**	System can't depend on **`/product`**. Not allowed.
`/product` → **`/vendor`**	❌ **No access**	Strongly restricted by SELinux and ABI policies
`/vendor` → **`/product`**	❌ **No access**	Vendor partition must boot even if **`/product`** is missing
`/system` → **`/vendor`**	✅ (via stable VNDK)	Only through declared VNDK
`/vendor` → **`/system`**	✅ (VNDK only)	Only through VNDK — stable ABI
`/system` → **`/system_ext`**	✅ Yes	system_ext is part of core
`/product` → **`/system_ext`**	✅ Limited	Same rules as accessing **`/system`** (only product-available modules)

✅ Summary:

- ✅ **`/product` can depend on `/system`** — but only **safe, allowed APIs** (like VNDK, public/system APIs).
- ❌ **`/system` must not depend on `/product`** — to ensure system is **modular and bootable alone**.
- ❌ **`/product` ↔ `/vendor`** interaction is strictly **prohibited** and blocked via **SELinux** and **build rules**.

💡 Why Not Bidirectional?

Because of **modularity and updatability**:

1. **`/system`** must remain **generic and reusable** across many devices (hence no reliance on **`/product`**)
2. **`/vendor`** must boot independently without **`/product`**
3. **`/product`** may be updated with OTA (or even removed), so can't be critical to core boot

---

🛡️ Enforcement Ensures:

- **`/product`** is **safe**, **optional**, and **replaceable**
- **`/system`** remains **stable**
- **`/vendor`** is **independent** and reliable

Would you like a **visual diagram** summarizing these directional rules?

42. can you explain this enforcement in simple words with examples

Absolutely! Let me explain **interface enforcement** between partitions in **very simple words**, using **real-life analogies and examples**:

🧠 What Is "Interface Enforcement"?

It's like setting **rules about who can talk to whom** and **what they are allowed to say**.

In Android, there are different **partitions** like:

- `/system`: Core Android (like a shared library)
- `/product`: Custom OEM features (like extra toppings)
- `/vendor`: SoC-specific code (like device hardware drivers)

Interface enforcement = Who can use whose functions/libraries — with **strict rules**, to avoid problems during updates.

🍕 Real-Life Analogy

Imagine a **pizza shop**:

- The **kitchen (system)** makes the standard base pizzas.
- The **franchise owner (product)** adds special toppings for different countries.
- The **supplier (vendor)** provides dough, ovens, and cheese machines (hardware).

Interface enforcement rules:

Who	Can ask for help from	Rule
Kitchen (system)	Franchise owner (product)	❌ **No!** Kitchen can't rely on local toppings. It must work even if toppings aren't available.
Franchise owner (product)	Kitchen (system)	✅ **Yes!** Can use base pizzas, but only the ones marked “safe to use” (like plain margherita).
Franchise owner (product)	Supplier (vendor)	❌ **No!** Can't directly talk to cheese machine supplier. Not allowed.
Supplier (vendor)	Kitchen (system)	✅ **Yes!** But only use officially supported kitchen tools (VNDK only).

📱 Android Example (Code-level)

Let’s say:

- `libcamera.so` is in `/system` → It is **VNDK** (safe public library)
- `libcustomfeatures.so` is in `/product`
- `libdriver_isp.so` is in `/vendor`

🔒 Enforcement in Action:

1. ✅ App in `/product` **can use** `libcamera.so` (if marked `product_available: true`)
2. ❌ Binary in `/system` **CANNOT use** `libcustomfeatures.so`
3. ❌ HAL in `/vendor` **CANNOT use** `libcustomfeatures.so` from `/product`
4. ✅ HAL in `/vendor` **CAN use** `libcamera.so` from `/system` (if it’s a VNDK lib)

💥 What Happens If You Break the Rule?

If you try to use a disallowed function:

- 🔧 **During build**: You get a build **error** — like `cannot link to non-public API`
- 🚫 **During boot/runtime**: The process may **crash** with error like:

```objectivec
CANNOT LINK EXECUTABLE "xyz": library "libabc.so" not found
```

🔑 Why Is This Important?

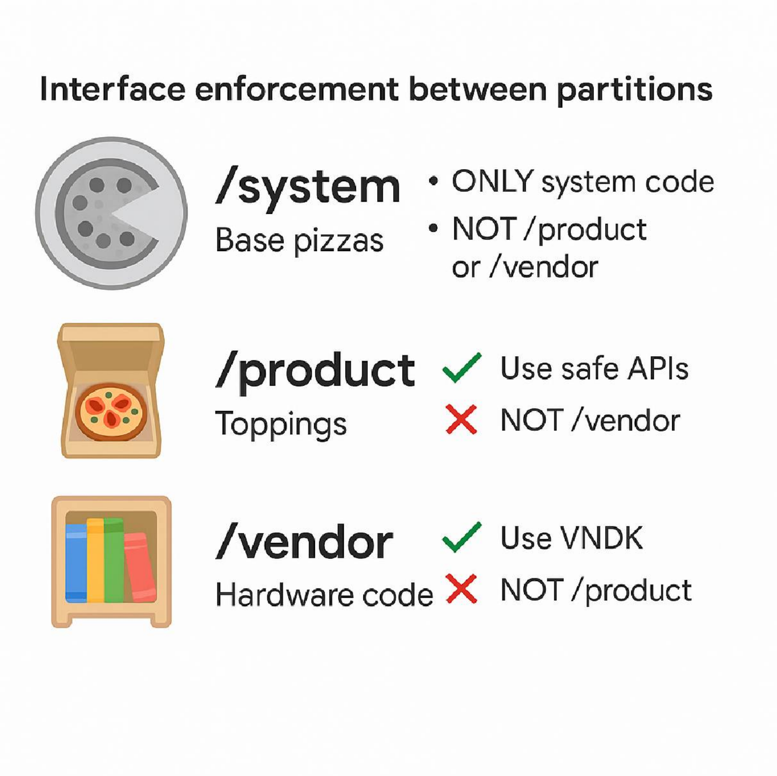
- 🧩 Keeps **modularity** — each piece can be updated separately.
- 🚀 Makes OTA updates and **Dynamic Partitions** possible.
- 🔐 Ensures **security** and stability by only using known, stable interfaces.

✅ TL;DR

Partition	Can Access
`/system`	ONLY system code, NOT `/product` or `/vendor`
`/product`	Can use **safe APIs** from `/system`, NOT from `/vendor`
`/vendor`	Can use **VNDK** from `/system`, NOT from `/product`

Would you like me to generate a diagram showing all this in simple arrows and icons?

43. ↳
step-by-step checklist to apply enforcement on your own custom device.
yes please



Here's a **step-by-step checklist** to apply **native and Java interface enforcement** for the **product partition** on your **custom Android device** (from Android 11+):

✅ Step-by-Step Enforcement Checklist

🔹 [1] Enable Native Interface Enforcement

Goal: Restrict native libraries/binaries in `/product` so they only use approved interfaces.

✅ **In `device.mk` or product config:**

```makefile
PRODUCT_PRODUCT_VNDK_VERSION := current
```

- This enables enforcement for native linking (bionic linker + Soong).
- Automatically enabled if shipping API level > 29.

🔹 [2] Fix Build-Time Errors (Native)

- 🔍 Check for build errors like:
  - `missing product variant`
  - `cannot link to non-product module`
- 🔧 Fixes:
  - Add `product_available: true` in the `Android.bp` of shared libraries:

```python
cc_library {
  name: "mylib",
  product_available: true,
  ...
}
```

- Use `product_specific: true` if the module is strictly for `/product`.

🔹 [3] Flash the Build & Boot Device

- Flash the device and **watch for logs**:

  ```arduino
  CANNOT LINK EXECUTABLE: ... violates linker config
  ```

- Fix any **runtime link failures** by:
  - Adding missing dependencies in `shared_libs`
  - Moving binaries/libraries to the correct partition

🔹 [4] Enable Java Interface Enforcement

Goal: Restrict apps/libraries in `/product` to use only public and @SystemApi Java APIs (no hidden or private APIs).

✅ **In `device.mk`:**

```makefile
PRODUCT_ENFORCE_PRODUCT_PARTITION_INTERFACE := true
```

- Automatically enabled if shipping API level > 29.

🔹 [5] Fix Build-Time Errors (Java)

- 🔍 Look for errors like:
  - Linking to hidden/private API
  - Using `@hide` methods
- 🔧 Fixes:
  - Use `sdk_version: "system_current"` or `sdk_version: "current"` in Android.bp
  - Replace hidden APIs with public/system APIs

🔹 [6] Verify Runtime Behavior

- 🧪 Enable runtime checks for hidden API access using:
  - **Logcat**: `StrictMode.detectNonSdkApiUsage()`
  - **Veridex Tool**:

    ```bash
    ./art/tools/veridex/appcompat.sh --dex-file=myapp.apk
    ```

🔹 [7] Enforce Artifact Path Separation (Optional)

Goal: Ensure only system modules go into `/system`, others to `/product`, `/vendor`, etc.

✅ **In `device.mk`:**

```makefile
PRODUCT_ENFORCE_ARTIFACT_PATH_REQUIREMENTS := strict
```

✅ **Define allowed install paths:**

```makefile
_my_paths := $(TARGET_COPY_OUT_ROOT)/ $(TARGET_COPY_OUT_SYSTEM)/
$(call require-artifacts-in-path, $(_my_paths),)
```

🔹 [8] Finalize Enforcement

- Make sure:
  - All native and Java interface violations are resolved.
  - Build is clean and boots correctly.
  - Runtime logs are clean of linker/hidden API violations.

🔐 Outcome

- ✅ Clean separation of system ↔ product interfaces
- ✅ Secure OTA updates with stable interfaces
- ✅ Modular code with maintainable boundaries

Let me know if you want a template `device.mk` or example `Android.bp` setups!
Sure! Let's explain **interface enforcement** (like `PRODUCT_PRODUCT_VNDK_VERSION := current`) in **very simple words** with examples.

🎯 What is "interface enforcement" in Android?

Think of each **partition** like a room in a house:

- `/system` is the **main room** with Android OS
- `/vendor` is the **SoC/chip room** (for hardware stuff)
- `/product` is the **OEM room** (your device-specific apps/libraries)
- `/system_ext` is a shared room for both Google & OEM code

Now:
👉 Interface enforcement means:
"What each room is allowed to talk to" is strictly controlled.

🧱 Native Interface Enforcement

🔧 Flag:

```makefile
PRODUCT_PRODUCT_VNDK_VERSION := current
```

✅ What it does:

- Only lets code inside `/product` use **safe, public libraries**.
- Prevents `/product` code from secretly using system internals.

🧠 Why?

So OEM features don't break when Android updates.
Keeps things stable and secure.

✅ Example:

✅ Allowed:

A C++ binary in `/product/bin` wants to use a system library:

```cpp
#include <log/log.h>   // OK – it's a public VNDK library
```

❌ Not Allowed:

The same binary tries to use a private system function:

```cpp
#include "internal_power.h"  // ❌ Boom! Build error!
```

☕ Java Interface Enforcement

🔧 Flag:

```makefile
PRODUCT_ENFORCE_PRODUCT_PARTITION_INTERFACE := true
```

✅ What it does:

- Apps in `/product` can only use public and system APIs.
- No access to **hidden** or **@hide** APIs.

✅ Example:

✅ Allowed:

```java
PowerManager pm = (PowerManager) context.getSystemService(Context.POWER_SERVICE);
pm.isInteractive();  // OK
```

❌ Not Allowed:

```java
pm.reboot("recovery");  // ❌ This is hidden API → Not allowed
```

If you try this, build will fail or app will crash.

🔒 Summary in Simple Words

Partition	Can Access (With Enforcement)	Can't Access
`/product`	Public/system APIs and libraries	Hidden/private stuff in system/vendor
`/vendor`	Only VNDK/public APIs	Internal system APIs
`/system`	Can use almost everything (but avoid hidden APIs)	—

If you want, I can show a visual diagram or a real Android.mk/BP file example too!

44. ⤷
🔹 **\[7\] Enforce Artifact Path Separation (Optional)** **Goal**: Ensure only system modules go into `/system`, others to `/product`, `/vendor`, etc. #### ✅ In `device.mk`: makefile CopyEdit `PRODUCT_ENFORCE_ARTIFACT_PATH_REQUIREMENTS := strict` #### ✅ Define allowed install paths: makefile CopyEdit `_my_paths := $(TARGET_COPY_OUT_ROOT)/ $(TARGET_COPY_OUT_SYSTEM)/ $(call require-artifacts-in-path, $(_my_paths),)`
why this

Great question! Let's break down:

🔹 Why `PRODUCT_ENFORCE_ARTIFACT_PATH_REQUIREMENTS := strict` exists

This setting **strictly controls where** files (apps, binaries, libraries) get installed during the Android build.

🏠 Analogy

Imagine building a **hotel**. You have:

- 🛏️ `/system` = VIP floor (only system code lives here)
- ⚙️ `/vendor` = technician floor (hardware drivers)
- 🎁 `/product` = OEM floor (your brand's special features)
- 📦 `/system_ext` = shared floor (Google + OEM common apps)

You don't want a technician (from `/vendor`) sleeping in the VIP room (`/system`) — that's **a rule violation**!

✅ What this flag does:

```makefile
PRODUCT_ENFORCE_ARTIFACT_PATH_REQUIREMENTS := strict
```

It says:

> "Only files defined in this makefile can install to `/system`. If someone else tries to install there, stop the build!"

🛡️ Why enforce this?

1. **To protect partition boundaries**
   → Prevent `/product` or `/vendor` apps from leaking into `/system`
2. **Ensure updatability**
   → OTA can update system without breaking vendor/product apps
3. **Make builds modular & clean**
   → Clear ownership: system code in `system.mk`, product code in `product.mk`

🧠 How it works

In `device.mk`:

```makefile
PRODUCT_ENFORCE_ARTIFACT_PATH_REQUIREMENTS := strict
```

And define allowed install paths like:

```makefile
_my_paths := \
  $(TARGET_COPY_OUT_ROOT)/ \
  $(TARGET_COPY_OUT_SYSTEM)/

$(call require-artifacts-in-path, $(_my_paths),)
```

This tells the build system:

✅ Only install files defined in this makefile to `/` or `/system`.
❌ If any other makefile tries to install here — **fail the build**.

🔧 Example

If `vendor.mk` tries to install to `/system/bin/foo` — the build will crash:

> ❌ ERROR: "foo" not allowed to install into `/system`.

✅ Fix:
Move `foo` to `/vendor/bin` or define it in `system.mk`.

📌 Summary

Purpose	Benefit
Prevent cross-partition pollution	Safer, modular Android builds
Ensure OTA/update compatibility	System can be updated independently
Enforce clear boundaries	Clean, maintainable AOSP device trees

---

Let me know if you want a diagram or flow of this in action!

45. Trusty OS (TOS) partitions

bookmark_border
Trusty is Google's implementation of a Trusted Execution Environment (TEE) OS that runs alongside Android. This is the specification for devices using Arm Trustzone technology to provide a TEE. If your ARM devices use Trusty as the secure OS solution, implement the bootloader as described in the following sections.

Initialize TOS
To load and initialize the Trusty OS (TOS), a bootloader should:

Set up and configure all available RAM.
Initialize at least one serial port.
Verify signature of TOS image.
Load TOS into RAM (execution from flash or TCM isn't supported).
Jump to the first instruction in the TOS image after setting up the state and registers as described below.
Call in to the TOS image
Configure the following state at entry:

MMU turned off
Data cache flushed and turned off (instruction cache can be on or off)
All interrupts (IRQs and FIQs) disabled
CPU in SVC mode on ARMv7 and EL3 on ARMv8
Registers in the following state:
r0/x0: Size of memory allocated to TOS.
r1/x1: Physical address of a contiguous block of memory that contains platform-specific boot parameters. The layout of this block is platform-specific.
r2/x2: Size of the above block of memory.
r14/x30: Return address to jump to (in nonsecure mode) after TOS initializes.
Note: r0-r3/x0-x3 also serve as scratch registers to TOS. Don't expect their values to be preserved upon return.
On a 64-bit platform:

Only w0-w2 are used for parameters, so x0-x2 should contain only 32-bit values.
x30 can contain a 64-bit value.
The value in x0 when added to the base address of TOS entry-point should result in a 32-bit value. The same applies to the size in register x2 when added to the address of boot parameter block in x1.
Return from TOS
After TOS completes initialization, it returns to the bootloader in nonsecure mode (SCR.NS set to 1) so that the bootloader can continue loading the primary operating system (that is, Android).

Here's a simplified explanation of **Trusty OS (TOS) partitions** and why they matter:

🔐 What is Trusty OS (TOS)?

Trusty OS is a small, **secure operating system** made by Google.
It runs *alongside Android*, but in a **Trusted Execution Environment (TEE)** — an isolated, hardware-backed area for **secure tasks**.

It runs on **ARM TrustZone**, a hardware feature that splits the CPU into two "worlds":

- 🌍 **Normal World** → Android runs here
- 🔐 **Secure World** → Trusty OS runs here

🚀 What happens during boot?

When the device boots, the **bootloader** needs to:

1. 🧠 **Setup memory** (RAM)
2. 🛠️ **Init serial port** (for debugging/logging)
3. ✅ **Verify** the signature of the Trusty OS image (to ensure it hasn't been tampered)
4. 📦 **Load Trusty OS** into memory

5. 🔁 **Jump to it** with some registers set properly

🧭 What does the bootloader pass to Trusty?

Before jumping to Trusty OS, the bootloader passes:

Register	Contains
r0/x0	Size of memory allocated to Trusty OS
r1/x1	Address of boot parameters (platform-specific block)
r2/x2	Size of the boot parameter block
r14/x30	Address to return to (to continue loading Android)

Other settings include:

- MMU off
- Caches flushed
- Interrupts disabled
- CPU in secure supervisor mode (SVC or EL3)

This ensures **Trusty runs in complete isolation**.

🧩 Why Trusty OS needs a partition?

Devices usually store Trusty OS as a **separate binary/partition** in flash storage — just like they do for **`boot.img`** or **`vendor.img`**.

This is important because:

- 🔒 It's updated **independently**
- 🧪 It's **verified** separately during boot (signature check)
- 💻 It's **loaded before Android**

🔄 What happens after Trusty initializes?

After Trusty OS finishes setup:

1. It returns **control back to the bootloader**
2. The bootloader then boots Android (normal world)

🔍 What does Trusty OS do?

It handles:

- Biometric processing (e.g. fingerprint/face)
- Secure keys & keystore
- Digital Rights Management (DRM)
- Verified boot attestation
- Secure UI

All in isolation from Android — so even if Android is compromised, Trusty is still protected.

🧠 Summary

Topic	Meaning
Trusty OS	Secure OS running alongside Android on ARM TrustZone
Why partition	To store and verify Trusty OS separately
Bootloader's job	Setup RAM, verify & load Trusty OS, pass registers, jump

Topic	Meaning
Return flow	Trusty initializes, returns to bootloader → Android boots

---

Let me know if you want a **boot flow diagram** or a simplified step-by-step animation of how Trusty and Android boot together!

46. what's the job of Trusty OS explain in simple words

Here's a simple explanation of what **Trusty OS** does:

🔐 Trusty OS = Security Bodyguard Inside Your Phone

Think of your phone like a house:

- 🏠 Android = the main part of the house (apps, system, UI)
- 🚪 Trusty OS = a **secret, locked room** inside the house that only handles **very sensitive things**

🛡️ What is Trusty OS used for?

It handles tasks that must stay **private and secure**, even if Android gets hacked.

💳 1. Stores and protects secrets

- Encryption keys
- Passwords
- Fingerprint/face unlock data

> 🔒 These are never stored in Android. Trusty keeps them safe inside the secure world.

🧑‍💻 2. Runs secure apps (Trusted Applications)

Like:

- Fingerprint recognition
- Keystore
- Digital rights (DRM for Netflix etc.)

These apps run in Trusty OS instead of Android — so they are safe from malware.

📱 3. Handles secure UI

If you enter your PIN, pattern, or fingerprint:

- The screen shown is controlled by Trusty (not Android), so malware can't fake it or steal it.

🔄 4. Verifies Android during boot

Trusty helps with **Verified Boot**:

- It checks Android hasn't been modified or tampered with before letting it boot.

🔌 5. Communicates safely with hardware

Trusty talks to **secure hardware components**, like:

- Fingerprint reader
- Secure element (for payments)
- TPM (Trusted Platform Module, if available)

🔧 Who controls Trusty OS?

- It's **made by Google**.
- It runs **alongside Android**, but in a **completely separate, secure area** of the processor (ARM TrustZone).
- Android apps and processes **can’t access it**.

📦 Summary (in 1 line)

> **Trusty OS is a tiny, secure operating system that handles sensitive stuff like biometric data, cryptography, and secure boot — keeping it safe even if Android gets hacked.**

Would you like a **visual diagram** or a step-by-step boot flow showing Trusty OS in action?

47. which will be loaded first Trusty OS or Android

Trusty OS is loaded *before* Android.

🔄 Boot Sequence (Simple Explanation):

1. **Boot ROM** (built into hardware)
    - Very first code that runs in the device
    - Loads and runs the **bootloader**
2. **Bootloader**
    - Initializes memory, CPU, storage, etc.
    - Verifies and **loads Trusty OS** into the *secure world* (ARM TrustZone)
    - **Starts Trusty OS first** ✅
3. **Trusty OS**
    - Runs security initialization (like fingerprint key vaults, verified boot, etc.)
    - After it's ready, it **returns control** to the bootloader
4. **Android OS**
    - Bootloader now loads and boots the **Android Linux kernel**
    - Android runs in the *normal world* (outside TrustZone)

✅ Why Trusty loads first?

Because:

- Trusty protects Android.
- Android needs Trusty to verify it’s safe to boot.
- Android relies on secure keys/services set up by Trusty.

🎯 Summary:

> ✅ **Trusty OS is loaded first by the bootloader**
> 🔄 Then it hands control back to continue booting **Android OS**

Would you like a **flowchart diagram** for this boot order?

48. map the partitions with aosp code directories

Mapping **Android partitions** to their corresponding **AOSP code directories** helps you understand where in the Android source code each partition's content is defined or built from. Below is a clean, categorized **partition-to-AOSP-directory mapping**:

📁 Partition to AOSP Source Directory Mapping

Partition	Purpose	AOSP Directories / Build Targets
`/system`	Core OS: Frameworks, apps, configs, etc.	`frameworks/`, `packages/`, `system/`, `build/target/product/`
`/system_ext`	Extensions for system (OEM features, but not vendor-specific)	`packages/apps/`, `frameworks/`, `build/target/product/`
`/vendor`	SoC & board-specific binaries, HALs, SEPolicy	`vendor/<oem>/`, `hardware/`, `device/<oem>/<device>/`
`/odm`	ODM-specific HW features; board-level custom HALs	`device/<odm>/<device>/`, custom `Android.mk`, `.bp` files
`/product`	Product-specific features, apps, locales, configs	`product/`, `device/<oem>/<device>/product.mk`, etc.
`/oem` (legacy)	Factory-flashed branding, wallpapers, etc. (not updateable)	Prebuilt, often outside AOSP tree (flashed separately)
`/recovery`	Mini-Android OS for updates/factory reset	`bootable/recovery/`, `system/core/`, `external/`
`/boot`	Kernel + initramfs (ramdisk)	`kernel/`, `device/<oem>/<device>/`, `init/`, `system/core/`
`/dtbo`	Device Tree Overlays (used by kernel)	`device/<oem>/<device>/`, DT source in `kernel/`
`/vbmeta`	Verified Boot metadata	Generated from AVB configs: `build/make/tools/releasetools/`
`/super`	Container for dynamic partitions like system, vendor, etc.	`build/make/tools/releasetools/`, `device.mk`, `BoardConfig.mk`
`/data`	Runtime user data (not part of AOSP, created at runtime)	N/A – runtime partition
`/cache` (legacy)	OTA/cache use; deprecated on most modern devices	N/A – runtime only
`/metadata`	Stores rollback indexes, encryption metadata	Managed by update_engine; not built from AOSP directly
`/persist`	Stores calibration data, Wi-Fi MAC, etc.	N/A – typically prebuilt & flashed at factory
`/system_dlkm`	GKI kernel modules (dynamic partition for kernel .ko)	`BOARD_SYSTEM_KERNEL_MODULES`, modules from `kernel/`
`/vendor_dlkm`	Vendor kernel modules	`BOARD_VENDOR_KERNEL_MODULES`, modules from `vendor/`
`/init_boot` (A13+)	Contains ramdisk/init files separated from boot partition	`init/`, `system/core/init`, `device/<vendor>/<device>/`

🧩 Key Build Files to Know:

- `device/<vendor>/<device>/BoardConfig.mk`: Defines partition sizes and partition use.
- `device/<vendor>/<device>/device.mk`: Specifies what goes where (apps, libs, HALs).
- `product.mk`: Used for `/product` partition layout.
- `Android.bp` / `Android.mk`: Define install paths via `LOCAL_MODULE_PATH`, `product_specific`, etc.
- `build/target/product/` and `build/make/tools/`: Build rules and templates.

🛠️ Example: You add a native HAL

- If for a specific SoC: goes to `/vendor/lib[64]/hw/` → defined in `vendor/` or `hardware/`
- If for specific ODM board: goes to `/odm/lib[64]/hw/` → defined in `device/<odm>/...`

Would you like a **diagram** that shows this mapping visually?

49. why in some cases one partition has mapping to multiple directories?

Great question! A single partition in Android often maps to **multiple AOSP directories** because:

✅ Reason 1: Partition = Logical Output, Directories = Source Inputs

A **partition** like `/system` is an **output image** (e.g., `system.img`) built from **many different source folders** in AOSP that provide:

- apps
- framework code
- libraries
- configs
- scripts

→ So naturally, it pulls content from **multiple source paths**.

✅ Reason 2: Layered Architecture = Separation of Concerns

Different **layers** of Android are maintained in different directories. For example, the `/system` partition includes:

Component	Source Directory
Java framework	`frameworks/base/`
Core system apps (Settings etc.)	`packages/apps/`
Init scripts & configs	`system/core/`, `init/`
Default apps & overlays	`build/target/product/`

All of this ends up inside the `system.img` or `/system`.

✅ Reason 3: Partition May Be Composed Based on Build Configs

The `device.mk`, `BoardConfig.mk`, and `.bp/.mk` files **pull modules** from different places depending on:

- Whether they are **product**, **vendor**, or **system** specific
- Whether they're needed by **SoC**, **OEM**, or **Google**

So `/product` could include:

- Locale-specific RRO overlays → `product/overlay/`
- OEM custom apps → `device/<oem>/<device>/`
- Java libraries → `product/framework/`

✅ Reason 4: Some Sources are Shared Across Partitions

Example:
An app might be defined in `packages/apps/FooApp/`, but installed to:

- `/system/app/FooApp/` → if it's for all SKUs
- `/product/app/FooApp/` → if it's SKU-specific
- `/vendor/app/FooApp/` → if it's vendor-specific

The **source is the same**, but the **partition is selected by flags** like:

- `product_specific: true`
- `vendor: true`
- `LOCAL_MODULE_PATH := $(TARGET_OUT_PRODUCT_APPS)`

✅ Reason 5: Partition as Merge of Subsystems

Especially in `/vendor` and `/odm`, different SoC vendors (Qualcomm, MediaTek) and ODMs contribute different pieces:

Source Directory	Contributes to
`hardware/qcom/`	`/vendor` (HALs)
`vendor/<oem>/device/`	`/vendor` or `/odm`
`device/<oem>/<device>/`	ODM/board-specific configs

They all contribute to a single partition because that's where **board-level code** is grouped.

🧠 Analogy: Partition as a Cake, Directories as Ingredients

You can think of:

- A **partition** like `/system` as a **cake**.
- AOSP directories like `frameworks/`, `packages/`, and `system/` are the **flour, sugar, eggs, frosting**.

They all go into one final thing, but are **separate for modularity** and maintenance.

Would you like a tree diagram showing **which AOSP directories contribute to each partition**?

50. a same directory or any can belong to many partitions?

Excellent follow-up! Let's break it down clearly:

✅ Can a same AOSP directory contribute to multiple partitions?

👉 Yes, absolutely.

But **how** and **where** its contents land depends on the **build configuration flags** in `.mk` or `.bp` files — not on the directory itself.

✅ Example 1: App used in multiple partitions

Let's say we have this source directory:

```bash
packages/apps/Calculator/
```

This **can be installed to different partitions** depending on how it's built:

Build Flag or Rule	Target Partition
Default (no special flag)	`/system/app/`
`product_specific: true` or `LOCAL_PRODUCT_MODULE := true`	`/product/app/`
`vendor: true` or `LOCAL_VENDOR_MODULE := true`	`/vendor/app/`
`odm: true`	`/odm/app/`

✅ Example 2: Shared HAL

Let's say a HAL is written in:

```swift
hardware/interfaces/sensors/1.0/
```

This HAL may be:

- Used in `/vendor` for SoC-level implementations.
- Also needed in `/odm` for board-specific customization.

Again, the **same directory**, but the output is controlled by the `Android.bp` rules in the modules that use or wrap it.

✅ Why allow this flexibility?

Because Android is built for many devices, SKUs, and vendors:

- The **same app** or **library** might be **OEM-branded** and go in `/product`
- Might be used **universally** and go in `/system`
- Might be **SoC-specific** and go in `/vendor`

Rather than duplicating code, AOSP allows **reusing the same source directory**, but controlling where it ends up using flags.

✅ How does the build system know where to place it?

The partition is chosen by:

- `product_specific: true` → product
- `vendor: true` → vendor

- `odm: true` → odm
- `system_ext_specific: true` → system_ext
- Default → system

This is handled in:

- `Android.bp` for Soong
- `Android.mk` for legacy Make

✅ Summary

Question	Answer
Can a **same AOSP directory** go to many partitions?	✅ Yes
Who decides which partition it goes to?	⚙️ Build rules (`.mk` or `.bp`)
Is code copied or reused?	🔁 Reused – compiled differently if needed
Why is this useful?	📱 OEM/SKU/vendor flexibility

Let me know if you want a **visual example** (like a tree or table) showing how one app/library can map to different partitions!

51. how to map these directories to these partitions
tell for all partitions
please create table for this

Here's a **complete table** that maps **Android AOSP source directories** to their typical **output partitions**. This helps you understand **which partition** a module (app, HAL, lib, etc.) ends up in based on **where it is in AOSP** and **what flags it uses**.

📦 Partition-to-Directory Mapping Table (AOSP)

Partition	Purpose	Mapped Directories (Typical Sources)	Install Flag (Soong/Make)
`/system`	Generic OS core, shared across all devices	- `frameworks/` - `packages/apps/` - `system/` - `libcore/` - `bionic/`	*Default* (no special flag)
`/system_ext`	OEM extensions to system (but shared across product variants)	- `packages/apps/` (OEM apps) - `frameworks/` - any app/lib marked as `system_ext_specific: true`	`system_ext_specific: true`
`/product`	Product-specific customizations (e.g., carrier or locale features)	- `packages/apps/` (product apps) - `frameworks/` - overlays, XML, media, build.prop - `product/` directory	`product_specific: true` or `product_available: true`
`/vendor`	SoC-specific components from chipset vendor	- `hardware/` - `vendor/` - proprietary blobs - `device/` (customized vendor.mk)	`vendor: true` or `vendor_available: true`
`/odm`	Board/device-specific code from ODMs (OEM hardware customizations)	- `device/<vendor>/<board>/` - customized HALs, RROs, `.rc` files, `.ko` files	`odm: true` or `LOCAL_ODM_MODULE := true`
`/boot`	Kernel + ramdisk (early init)	- `kernel/` - ramdisk stuff from `init/`, `system/core/init/`, `rootdir/`	`BOARD_KERNEL_...` configs in BoardConfig.mk
`/dtbo`	Device Tree Overlays	- `device/.../*.dts`	`BOARD_PREBUILT_DTBOIMAGE` or `BOARD_KERNEL_...` options
`/vbmeta`	Verified Boot metadata	- Signing metadata — not directly from a source dir	`BOARD_AVB_...` configs
`/recovery`	Recovery OS image	- `bootable/recovery/`	Automatically included in recovery image
`/data`	Runtime user data (apps, cache, etc.)	❌ **No source mapping** — created at runtime	❌ (Not built from AOSP)
`/system_dlkm`	GKI (Generic Kernel Image) kernel modules partition (Android 13+)	- `kernel/prebuilts/` - `.ko` files in staging folder	`BOARD_SYSTEM_KERNEL_MODULES`, `BOARD_SYSTEM_DLKM_SRC`
`/vendor_dlkm`	Vendor-specific kernel modules partition (Android 13+)	- Device-specific `.ko` modules	`BOARD_VENDOR_KERNEL_MODULES`, `vendor: true`

🔍 Example: Same Source, Different Output

Suppose `packages/apps/Calculator/` has:

- `product_specific: true` → goes to `/product/app/Calculator/`
- `vendor: true` → goes to `/vendor/app/Calculator/`
- no flag → goes to `/system/app/Calculator/`

So **source directory ≠ fixed partition** — the **install flags and makefiles decide**.

⚠️ Why multiple directories can map to one partition?

Because many modules from different areas (apps, HALs, libs) are compiled into the same partition image. For example:

- `/vendor` may include HALs from `hardware/interfaces/` **and** blobs from `vendor/`
- `/system` has Java libs from `libcore/`, native code from `bionic/`, apps from `packages/apps/`, etc.

Would you like a **color-coded chart** or a **flow diagram** showing this graphically?