Bootloader

1. Bootloader overview bookmark_border A bootloader is a vendor-proprietary image responsible for bringing up the kernel on a device. The bootloader guards the device state and is responsible for initializing the Trusted Execution Environment (TEE) and binding its root of trust. The bootloader also verifies the integrity of the boot and recovery partitions before moving execution to the kernel. Example bootloader flow Here's an example bootloader flow: Load and initialize memory. Verify the device according to Verified Boot flow. Verify the boot partitions, including boot, dtbo, init_boot, and recovery, according to the Verified Boot flow. As part of this step, check the boot image header version and parse the header accordingly. If A/B. 2
2. KASLRexplain in simple words with example. 5
3. Kernel Command Line Sourcesexplain simple words w. 7
4. Canonical boot reason bookmark_border Android 9 includes the following changes to the bootloader boot reason specification. Boot reasons A bootloader uses uniquely-available hardware and memory resources to determine why a device rebooted, then communicates that determination by adding androidboot.bootreason=<reason> to the Android kernel command line for its launch. init then translates this command line to propagate to the Android property bootloader_boot_reason_prop (ro.boot.bootreason). For devices launching with Android 12 or higher, using kernel version 5.10 or greater, androidboot.bootreason=<reason> is added to bootconfig instead of the kernel command line. Boot reason specifications Previous releases of Android specified a boot. 9
5. Boot image header bookmark_border Android 9 introduced a version field in the boot image header, enabling updates to the header while maintaining backward compatibility. The bootloader must check the header version field and parse the header accordingly. Devices launching with: Android 13 can use boot header version 3 or 4. For devices supporting the Generic Kernel Image (GKI) architecture, version 4 is the primary boot image and the os_version field in the boot header must be zero. The device bootloader is expected to obtain the version information from the Android Verified Boot (AVB) properties instead. Android 12 can use boot header version 3 or 4. For devices supporting the Generic Kernel Image (GKI) architecture, version 4 is the pri. 14
6. Implement bootconfig in Android 12 bookmark_border In Android 12, the bootconfig feature replaces the androidboot.* kernel cmdline options in use with Android 11 and lower. The bootconfig feature is a mechanism for passing configuration details from the build and bootloader to Android 12. This feature provides a way to separate the configuration parameters for Android user-space from those for the kernel. Moving the lengthy androidboot.* kernel parameters to the bootconfig file creates space on the kernel cmdline and makes it available for future expansion. Both the kernel and the Android user-space must support bootconfig. First release that has this support: Android 12 First kernel version that has this support: 12-5.4.xx kernel Imple. 20
7. androidboot.slot_suffix=_awhy we use this. 25
8. DTB images bookmark_border Android implementations can include a device tree blob (DTB) image for use by the bootloader. The location of the DTB image (and the options for specifying DTB image parameters) differs between Android releases. In Android 11, devices using the Generic Kernel Image (GKI) must support the vendor boot partition, which includes all vendor-specific information that was relocated from the boot partition. Because the DTB image contains vendor-specific data, it's now part of the vendor boot partition. To specify DTB image parameters, see Vendor boot header. In Android 10, devices can include the DTB image in the boot partition. To specify DTB image parameters, see Including the DTB image in the boot image. In Android. 27
9. Implement OTA updates bookmark_border To implement the over-the-air (OTA) updates, the bootloader must be able to access a recovery RAM disk during boot. If the device uses an unmodified AOSP recovery image, the bootloader reads the first 32 bytes on the misc partition; if the data there matches boot-recovery, the bootloader boots into the recovery image. This method enables any pending recovery work (for example, applying an OTA or removing data) to continue to completion. For details on the content of a block in flash used for communications by recovery and the bootloader, refer to bootable/recovery/bootloader_message/bootloader_message.h. Devices with A/B updates To support OTA updates on devices that use A/B updates, ensure that the. 30
10. Implement OTA updates bookmark_border To implement the over-the-air (OTA) updates, the bootloader must be able to access a recovery RAM disk during boot. If the device uses an unmodified AOSP recovery image, the bootloader reads the first 32 bytes on the misc partition; if the data there matches boot-recovery, the bootloader boots into the recovery image. This method enables any pending recovery work (for example, applying an OTA or removing data) to continue to completion. For details on the content of a block in flash used for communications by recovery and the bootloader, refer to bootable/recovery/bootloader_message/bootloader_message.h. Devices with A/B updates To support OTA updates on devices that use A/B updates, ensure that the. 33
11. ⚙️ Non-A/B OTA Update Flow -------------------------- ### 📌 1. **Recovery Partition Required** * Recovery image contains logic to: * Flash system/radio/vendor partitions. * Read update package from `/cache` or `/data`. ### 📌 2. **Bootloader Requirements** * Must boot into **recovery** when needed. * Typically reads the **first 32 bytes of `misc` partition**: * If string is `boot-recovery`, then boot recovery image instead of system. cpp CopyEdit `// bootloader_message.h defines: char command[32]; // e.g., "boot-recovery"` ### 📌 3. **Radio Image Updates** Handled via: * **Bootloader** (preferred): Flash radio directly, then reboot to recovery. * **Recove. 37
12. Retry counexplain in simple words. 39
13. Lock and unlock the bootloader bookmark_border By default, most Android devices ship with a locked bootloader, meaning that users can't flash the bootloader or device partitions. If needed, you (and device users with Developer options enabled) can unlock the bootloader to flash new images. Get into the bootloader To issue fastboot commands, you must be in the bootloader mode. One way to do this is to send the adb command adb reboot bootloader. Some phones also allow you to reboot into the bootloader by pressing a key combination (commonly volume down) while booting the device. Unlock the bootloader Once in the bootloader mode, to unlock the bootloader and enable partitions to be reflashed, run the fastboot flashing unlock command on the. 41
14. Move fastboot to userspace bookmark_border Fastboot is the name of a bootloader module and mode. Android 10 and higher supports resizable partitions by relocating the fastboot implementation from bootloader to userspace. This relocation enables moving the flashing code into a maintainable and testable common location with only the vendor-specific parts of fastboot implemented by a hardware abstraction layer (HAL). In addition, Android 12 and higher supports flashing ramdisks through an added fastboot command. Unify fastboot and recovery Because userspace fastboot and recovery are similar, you can merge them into one partition or binary. This provides advantages such as using less space, having fewer partitions overall, and having fastboot. 44
15. 🔁 How does bootloader enter `fastbootd`? -- 1. Bootloader writes `boot-fastboot` in BCB (boot control block). 2. Recovery boots and sees this command. 3. Recovery enters `fastbootd` mode.explain in simple words. 50
16. Generic Bootloader (GBL) overview bookmark_border The Generic Bootloader (GBL) is a standardized, updatable bootloader solution designed to streamline the Android boot process. GBL replaces the highly fragmented landscape of vendor-specific bootloaders with a single, consistently maintained component. This change reduces redundancy, improves security, and simplifies updates across the Android ecosystem. GBL consists of the following components: core Android boot logic The logic that includes the main program loop, boot mode detection, and loading the kernel. Fastboot A communication protocol and diagnostic tool used for device flashing and debug. Vendor extensions Extension of the aforementioned components of GBL by creating your own pro. 51
17. UEFI protocolexplain in simple words. 53
18. Deploy GBL bookmark_border This page explains how to deploy the GBL binary. Boot firmware requirements To use GBL, the boot firmware must meet the following requirements: UEFI compliance. The firmware must implement and use the required UEFI protocols. The firmware must also allow for vendor-specific extensions using defined UEFI protocols. Security. The firmware must implement all aspects of Android Verified Boot (AVB), ensuring only authenticated images are loaded. Boot modes. The binary should be able to handle various boot modes, such as normal boot, recovery boot, and fastboot. Dynamic partitioning. The boot firmware must implement slot selection logic so that it supports reading the correct A/B boot slot and is compatible with d. 55
19. how GBL worked UEFI explain in simple words. 58
20. Now tell me full boot flow in android as per various changes in android over the time. 59

1. Bootloader overview

bookmark_border
A bootloader is a vendor-proprietary image responsible for bringing up the kernel on a device. The bootloader guards the device state and is responsible for initializing the Trusted Execution Environment (TEE) and binding its root of trust. The bootloader also verifies the integrity of the boot and recovery partitions before moving execution to the kernel.

Example bootloader flow
Here's an example bootloader flow:

Load and initialize memory.

Verify the device according to Verified Boot flow.

Verify the boot partitions, including boot, dtbo, init_boot, and recovery, according to the Verified Boot flow. As part of this step, check the boot image header version and parse the header accordingly.

If A/B updates are used, determine the current slot to boot.

Determine if recovery mode should be booted. For more information, see Supporting OTA Updates.

Load the boot images, such as boot.img, vendor_boot.img, init_boot.img, and other proprietary vendor boot images. These boot images contain the kernel and ramdisk images.

Load the kernel into memory as a self-executable compressed binary. The kernel decompresses itself and starts executing into memory.

Load ramdisks and the bootconfig section into memory to create initramfs.

Additional bootloader-related features
Following is a list of additional bootloader-related features that you can implement:

Device tree overlay (DTO). A device tree overlay lets the bootloader to support different hardware configurations. A DTO is compiled into a device tree blob (DTB) which is used by the bootloader.

Kernel image virtual address randomization. The bootloader supports randomizing the virtual address at which the kernel image is loaded. To randomize the address, set RANDOMIZE_BASE to true in the kernel config. The bootloader must provide entropy by passing a random u64 value in the /chosen/kaslr-seed device tree node.

Verified Boot. Verified Boot lets the bootloader to ensure all executed code comes from a trusted source.

Boot config. Boot config is available in Android 12 and higher and is a mechanism for passing configuration details from the build and bootloader to the operating system. Prior to Android 12, kernel command-line parameters with the prefix of androidboot are used.

Over-the-air (OTA) updates. Android devices in the field can receive and install OTA updates to the system, app software, and time zone rules. This feature has implications on your bootloader implementation. For general information on OTA, see OTA updates. For bootloader-specific OTA implementation details, see Supporting OTA updates.

Version binding. Version binding binds security keys to the operating system and patch level version. Version binding ensures that an attacker who discovers a weakness in an old version of the system or the TEE software can't roll a device back to the vulnerable version and use keys created with the newer version. The bootloader must provide certain information to support version binding. For further information, see Version information in AVB properties.

Kernel command line
Concatenate the kernel command line from the following locations:

Bootloader command line: set of static and dynamic parameters determined by the bootloader

Device tree: from the chosen/bootargs node

defconfig: from CONFIG_CMDLINE

boot.img: from the command line (for offsets and sized, refer to system/core/mkbootimg/bootimg.h

As of Android 12, for androidboot.* parameters that we need to pass to Android userspace, we can use bootconfig instead of the kernel command line.

Here's a simplified and structured **overview of the Android bootloader**, its **flow**, and **key features**, explained in **simple terms**:

🔧 What is a Bootloader?

The **bootloader** is the **very first piece of code** that runs on your Android device when it's powered on. It is a **vendor-specific binary**, meaning each manufacturer provides their own version.

It performs **critical tasks before handing control over to the Android kernel**.

📋 Key Responsibilities of the Bootloader

1. **Initialize hardware and memory**
2. **Verify security and integrity** of boot images (using **Verified Boot**)
3. **Load the Android kernel and ramdisk**
4. **Setup Trusted Execution Environment (TEE)**
5. **Support A/B (Seamless) OTA Updates**
6. **Pass configuration to the OS**
7. **Randomize kernel load address (KASLR) for security**

🔄 Example Bootloader Flow

Here's how it generally works:

Step	Description
1️⃣	**Initialize memory** and setup low-level hardware
2️⃣	**Verify device state** using Verified Boot (check if bootloader is locked, etc.)
3️⃣	**Verify boot-related partitions** like **`boot.img`**, **`dtbo.img`**, **`init_boot.img`**, **`recovery.img`**
4️⃣	If using **A/B system**, pick the correct **slot (A or B)**
5️⃣	If a key combination is pressed, go to **Recovery Mode**
6️⃣	**Load kernel** (Linux compressed binary) into memory
7️⃣	**Load ramdisk** and **`bootconfig`** into memory to form **initramfs**
8️⃣	Transfer execution to the **Linux kernel** (it decompresses and runs)

🧩 Additional Features Bootloader Can Support

Feature	Description
✅ **Device Tree Overlay (DTO)**	Helps support multiple hardware variants by patching the base device tree with overlays at runtime
🔐 **Verified Boot (AVB)**	Ensures the code being booted is from a trusted source and not tampered
🎲 **KASLR (Kernel Address Space Layout Randomization)**	Bootloader provides random seed to load the kernel at random address for security
🛠️ **Bootconfig**	From Android 12+, configuration parameters passed via a special **`bootconfig`** section (instead of **`androidboot.*`** on kernel cmdline)
📦 **OTA Update Support**	A/B partitions or recovery flow requires bootloader support for update logic
🔑 **Version Binding**	Ensures rollback attacks can't be performed (older vulnerable OS/TEE can't be used with newer keys)

📝 Kernel Command Line Sources

The **kernel command line** is a string of parameters passed to the Linux kernel. It is constructed from:

Source	Notes
🔧 **Bootloader**	Adds dynamic or static flags like **`androidboot.verifiedbootstate=green`**

Source	Notes
🌲 **Device Tree** (`/chosen/bootargs`)	May contain hardware-specific boot arguments
⚙️ **Kernel defconfig** (`CONFIG_CMDLINE`)	Default command line from kernel source
📦 **boot.img**	Contains command line data passed during boot image creation

👉 **From Android 12+, use `bootconfig`** to pass **`androidboot.*`** properties cleanly to Android.

🔐 Trusted Boot Summary

- **First thing loaded**: Bootloader
- **Next**: Verifies signatures of kernel and ramdisk
- **TEE initialized early**
- **If bootloader is locked**, only verified images can run

Let me know if you want diagrams or a breakdown of **A/B OTA logic**, **boot.img format**, or **Verified Boot process** in detail.

> 2. ↪
> KASLR
> explain in simple words with example

Sure! Let's explain **KASLR** (Kernel Address Space Layout Randomization) in **very simple words**, with an **example**.

🔐 What is KASLR?

KASLR stands for:

> **Kernel Address Space Layout Randomization**

It's a **security feature** that helps protect your Android (or Linux) device from hackers.

🧠 Why is KASLR used?

Normally, when the device boots up, the **kernel** is loaded into memory at a **fixed address**. If an attacker knows this memory address, they can write malware that targets it directly.

KASLR solves this problem by loading the kernel into a **different, random memory address each time** the device boots.

This way, **even if someone tries to attack the kernel**, they **don't know where it is**, making attacks much harder.

📦 How does it work?

1. The **bootloader** generates a **random number** (called the **KASLR seed**).
2. It stores this number in the **device tree**, at `/chosen/kaslr-seed`.
3. When the kernel starts, it reads this seed and uses it to decide **where to load itself in memory**.
4. Result: kernel loads at a **new random address** on every boot.

📱 Simple Example

Imagine the kernel is like a **house**, and memory is like a **huge street** with many house numbers.

Without KASLR:

- The house (kernel) is **always at 123 Main Street**.
- Bad guys know this and can walk straight in.

With KASLR:

- Each time you start the device, the house **moves to a random new address** like:
  - Today: 817 Main Street
  - Tomorrow: 394 Main Street
- Bad guys don't know the address anymore — much harder to attack!

📋 In Bootloader Config

To enable KASLR:

- In the **kernel config** file, enable:

```ini
CONFIG_RANDOMIZE_BASE=y
```

- The **bootloader** must pass the random seed like this (in device tree):

```dts
chosen {
    kaslr-seed = <0xdeadbeef12345678>;
};
```

✅ Benefit

- **Makes it very hard** for attackers to predict where the kernel is.
- Stops many **memory-based attacks** (like buffer overflows or ROP).

Let me know if you want a diagram or how to check if KASLR is working on your Android device.

3. ↪
Kernel Command Line Sources
explain simple words w

Sure! Let's explain the **Kernel Command Line Sources** in **very simple words with an example**.

🧠 What is the Kernel Command Line?

When your Android phone boots up, the **bootloader** starts the **Linux kernel**.
To help the kernel know **how to behave**, the bootloader passes some **instructions** — this is called the **kernel command line**.

Think of it like giving the kernel a **note** that says:

> "Hey kernel, here's how you should start things up."

📦 Where does this command line come from?

The full **kernel command line** is created by combining pieces from **4 different places**:

Source	What it is	Example
1️⃣ Bootloader	The bootloader adds custom text	`androidboot.verifiedbootstate=green`
2️⃣ Device Tree	In the `chosen/bootargs` node	`console=ttyS0`
3️⃣ Kernel Config (`defconfig`)	Default settings in kernel code	`CONFIG_CMDLINE="console=tty0"`
4️⃣ boot.img	Image file containing kernel + cmdline	Custom settings passed during build

📋 Example Full Command Line

Let's say:

- Bootloader adds: `androidboot.verifiedbootstate=green`
- Device Tree has: `console=ttyS0`
- Kernel config has: `CONFIG_CMDLINE="loglevel=4"`
- boot.img adds: `root=/dev/mmcblk0p22 rw`

Then the **final command line** passed to the kernel looks like:

```ini
androidboot.verifiedbootstate=green console=ttyS0 loglevel=4 root=/dev/mmcblk0p22 rw
```

🤔 Why is this important?

These **parameters** tell the kernel:

- What device to boot from (`root=/dev/...`)
- What logs to show (`loglevel=4`)
- How to display output (`console=ttyS0`)
- Boot status (`androidboot.verifiedbootstate=green`)

📢 From Android 12+

Instead of sending these as plain text on the command line, Android now uses a **better way** called:

> `bootconfig`

So, for parameters like `androidboot.*`, we put them in a separate section of memory called `bootconfig`, making things cleaner and easier to manage.

✅ Summary

- Kernel needs **instructions** when it starts → passed as **command line**.
- These instructions come from **4 places** (bootloader, DT, defconfig, boot.img).
- They are **combined together** to form the final message.
- From Android 12+, we prefer using **bootconfig** instead of `androidboot.*` in cmdline.

Let me know if you want a diagram or a real Android example (`cat /proc/cmdline`).

4. Canonical boot reason

bookmark_border
Android 9 includes the following changes to the bootloader boot reason specification.

Boot reasons
A bootloader uses uniquely-available hardware and memory resources to determine why a device rebooted, then communicates that determination by adding androidboot.bootreason=<reason> to the Android kernel command line for its launch. init then translates this command line to propagate to the Android property bootloader_boot_reason_prop (ro.boot.bootreason). For devices launching with Android 12 or higher, using kernel version 5.10 or greater, androidboot.bootreason=<reason> is added to bootconfig instead of the kernel command line.

Boot reason specifications
Previous releases of Android specified a boot reason format that used no spaces, was all lowercase, included few requirements (such as for reporting kernel_panic, watchdog, cold/warm/hard), and which made allowances for other unique reasons. This loose specification resulted in the proliferation of hundreds of custom (and sometimes meaningless) boot reason strings, which in turn led to an unmanageable situation. As of the current Android release, the sheer momentum of near unparseable or meaningless content filed by the bootloader has created compliance issues for bootloader_boot_reason_prop.

With the Android 9 release, the Android team recognizes that the legacy bootloader_boot_reason_prop has substantial momentum and cannot be re-written at runtime. Any improvements to the boot reason specification must therefore come from interactions with bootloader developers and tweaks to the existing system. To that end the Android team is:

Engaging with bootloader developers to encourage them to:
Provide canonical, parseable, and recognizable reasons to bootloader_boot_reason_prop.
Participate in the system/core/bootstat/bootstat.cpp kBootReasonMap list.
Adding a controlled and runtime-rewritable source of the system_boot_reason_prop (sys.boot.reason). A limited set of system apps (such as bootstat and init) can rewrite this property, but all apps can be granted sepolicy rights to read it.
Informing users of the boot reason to wait until after userdata is mounted before trusting the content in the system boot reason property system_boot_reason_prop.
Why so late? While bootloader_boot_reason_prop is available early on in boot, it is blocked by the Android security policy on an as-need basis because it represents inaccurate, unparseable, and noncanonical information. In most situations, only developers with deep knowledge of the boot system should need to access this information. A refined, parseable, and canonical API for boot reason with system_boot_reason_prop can be reliably and accurately picked up only after userdata has mounted. Specifically:

Before userdata has mounted, system_boot_reason_prop will contain the value from bootloader_boot_reason_prop.
After userdata has mounted, system_boot_reason_prop may be updated to be compliant or to report more accurate information.
For this reason, Android 9 extends the period of time before the boot reason can be officially acquired, changing it from being immediately accurate in boot (with bootloader_boot_reason_prop) to being available only after userdata has mounted (with system_boot_reason_prop).

Bootstat logic depends on a more informative and compliant bootloader_boot_reason_prop. When that property uses a predictable format, it improves the accuracy of all controlled reboot and shutdown scenarios, which in turn refines and expands the accuracy and meaning of system_boot_reason_prop.

Canonical boot reason format
The canonical boot reason format for bootloader_boot_reason_prop in Android 9 uses the following syntax:

<reason>,<subreason>,<detail>…
Formatting rules:

Lower case
No blanks (use underline)
All printable characters
Comma-separated reason, subreason, and one or more detail instances.
A required reason that represents the highest priority reason why the device had to reboot or shutdown.
An optional subreason that represents a short summary of why the device had to reboot or shutdown (or who rebooted or shutdown the device).
One or more optional detail values. A detail may point to a subsystem to aid in determining which specific system resulted in the subreason. You can specify multiple detail values, which should generally follow a hierarchy of importance. However, it's also acceptable to report multiple detail values of equal importance.
An empty value for bootloader_boot_reason_prop is considered illegal (as this allows other agents to inject a boot reason after the fact).

Reason requirements
The value given for reason (first span, prior to termination or comma) must be of the following set divided into kernel, strong, and blunt reasons:

kernel set:
"watchdog"
"kernel_panic"
strong set:
"recovery"
"bootloader"
blunt set:
"cold". Generally indicates a full reset of all devices, including memory.
"hard". Generally indicates the hardware has its state reset and ramoops should retain persistent content.
"warm". Generally indicates the memory and the devices retain some state, and the ramoops (see pstore driver in kernel) backing store contains persistent content.
"shutdown"
"reboot". Generally means the ramoops state is unknown and the hardware state is unknown. This value is a catchall as the cold, hard, and warm values provide clues as to the depth of the reset for the device.
Bootloaders must provide a kernel set or a blunt set reason, and are strongly encouraged to provide a subreason if it can be determined. For example, a power key long press that may or may not have ramoops backup would have the boot reason "reboot,longkey".

No first-span reason can be part of any subreason or detail. However, because kernel set reasons cannot be produced by user space, "watchdog" may be reused after a blunt set reason, along with a detail of the source (for example, "reboot,watchdog,service_manager_unresponsive", or "reboot,software,watchdog").

Boot reasons should not require expert internal knowledge to decipher and/or should be human readable with an intuitive report. Examples: "shutdown,vbxd" (bad), "shutdown,uv" (better), "shutdown,undervoltage" (preferred).

Reason-subreason combinations
Android reserves a set of reason-subreason combinations that should not be overloaded in normal usage but can be used on a case-by-case basis if the combination accurately reflects the associated condition. Examples of reserved combinations include:

"reboot,userrequested"
"shutdown,userrequested"
"shutdown,thermal" (from thermald)
"shutdown,battery"
"shutdown,battery,thermal" (from BatteryStatsService)
"reboot,adb"
"reboot,shell"
"reboot,bootloader"
"reboot,recovery"
For more details, refer to kBootReasonMap in system/core/bootstat/bootstat.cpp and the associated git changelog history in the Android source repository.

Report boot reasons
All boot reasons, either from the bootloader or recorded in the canonical boot reason, must be recorded in the kBootReasonMap section of system/core/bootstat/bootstat.cpp. The kBootReasonMap list is a mix of compliant and legacy noncompliant reasons. Bootloader developers should register only new compliant reasons here (and shouldn't register noncompliant reasons unless the product has already shipped and cannot be changed).

Note: While system/core/bootstat/bootstat.cpp contains a kBootReasonMap section that lists a considerable number of legacy reasons, the presence of these reasons does not mean the reason string is approved for use. A subset of the list represents compliant reasons; we expect this subset to grow as bootloader authors register and explain additional compliant reasons.
We strongly recommend using existing, compliant entries in system/core/bootstat/bootstat.cpp and exercising restraint before using a noncompliant string. As a guideline, it is:

OK to report "kernel_panic" from the bootloader, as bootstat may be able to inspect ramoops for kernel_panic signatures to refine the subreasons into the canonical system_boot_reason_prop.
Not OK to report a noncompliant string in kBootReasonMap (such as "panic") from the bootloader, as this will ultimately break the ability to refine the reason.
For example, if kBootReasonMap contains "wdog_bark", a bootloader developer should:

Change to "watchdog,bark" and add to the list in kBootReasonMap.
Consider what "bark" means for those unfamiliar with the technology and determine if a more meaningful subreason is available.
Verify boot reason compliance
At this time, Android does not provide an active CTS test that can accurately trigger or inspect all possible boot reasons a bootloader could provide; partners can still attempt to run a passive test to determine compatibility.

As a result, bootloader compliance requires bootloader developers to voluntarily adhere to the spirit of the rules and guidelines described above.

We urge such developers to contribute to AOSP (specifically to system/core/bootstat/bootstat.cpp) and use this opportunity as a forum for discussions about boot reason issues.

Was this helpful?

Sure! Let's break this down in **very simple words** with **clear examples** so you understand what **Canonical Boot Reason** means in Android.

🔧 What is a "Boot Reason"?

A **boot reason** is the **cause** or **reason why your phone restarted or turned on**.

Example reasons:

- You pressed the power button.
- The phone crashed (kernel panic).
- It overheated and shut down.
- Battery was too low.
- A system update rebooted it.

🤖 Who sets this boot reason?

The **bootloader** — the first program that runs when your device turns on — decides **why** the phone is rebooting and passes this info to Android using a message like:

```bash
androidboot.bootreason=reboot,userrequested
```

This message is sent to Android via:

- Kernel command line (before Android 12)
- Bootconfig (from Android 12+)

📋 What is Canonical Boot Reason?

Canonical means: **standard**, **clear**, and **predictable** format.

Before Android 9, everyone wrote their own weird boot reasons like:

```arduino
"panic", "abc123", "powerbutton_hard_reset", "watchdogbark", "uv"
```

These were:

- Messy
- Hard to understand
- Not standardized

So Android 9 introduced a **standard (canonical) format** to make boot reasons:

- Clear for developers
- Easy to parse by the system
- Useful for logs and apps

✅ Canonical Format

General format:

```php-template
<reason>,<subreason>,<detail1>,<detail2>,...
```

Rules:

- All **lowercase**
- No spaces — use _ if needed
- Comma-separated parts
- `reason` is **mandatory**
- `subreason` and `details` are optional

🔍 Examples

Boot Reason	What It Means
`reboot,userrequested`	User manually restarted the phone
`shutdown,battery`	Phone shut down due to low battery
`shutdown,battery,thermal`	Phone shut down due to overheating caused by battery
`reboot,watchdog,system_server_unresponsive`	Phone restarted because the system became unresponsive
`kernel_panic,subsys,gpu`	Phone crashed due to GPU kernel panic

🔄 When Are These Used?

Android uses two boot reason properties:

Property	When Available	Where it comes from
`ro.boot.bootreason`	Early in boot	Set by the bootloader
`sys.boot.reason`	After data partition is mounted	Cleaned and canonical version

> Before Android 12, boot reason is passed via **kernel command line**.
> From Android 12+, it's passed via **bootconfig** instead.

🧠 Why Delay Canonical Reason Until Data is Mounted?

Because:

- Early boot reason (`ro.boot.bootreason`) can be messy or non-standard.
- Android waits until **data is mounted** to write a **clean, verified reason** in `sys.boot.reason`.

💡 What Bootloader Developers Must Do

- Use **standard reason names** like:
  - `reboot`
  - `shutdown`
  - `watchdog`
  - `kernel_panic`
- Optionally add **subreason/details** (like `userrequested`, `thermal`, etc.)
- Avoid using unknown or non-standard values (e.g., don't write `"panic"`, use `"kernel_panic"`)

They should also update `system/core/bootstat/bootstat.cpp` in AOSP so the system knows how to parse the new boot reason.

✅ Summary

Term	Meaning
`androidboot.bootreason`	Bootloader tells Android why the phone rebooted
Canonical format	Clear, standard format like **`shutdown,battery`**
`ro.boot.bootreason`	Early boot reason, from bootloader
`sys.boot.reason`	Cleaned up version, available later in boot
Goal	Make boot reasons easy to read, analyze, and use in apps

---

Let me know if you want:

- A list of valid reasons/subreasons
- How to check boot reason on your device
- Or a diagram of the boot process showing bootreason flow

5. Boot image header

bookmark_border
Android 9 introduced a version field in the boot image header, enabling updates to the header while maintaining backward compatibility. The bootloader must check the header version field and parse the header accordingly. Devices launching with:

Android 13 can use boot header version 3 or 4. For devices supporting the Generic Kernel Image (GKI) architecture, version 4 is the primary boot image and the os_version field in the boot header must be zero. The device bootloader is expected to obtain the version information from the Android Verified Boot (AVB) properties instead.
Android 12 can use boot header version 3 or 4. For devices supporting the Generic Kernel Image (GKI) architecture, version 4 is the primary boot image.
Android 11 can use boot header version 3. For devices supporting Generic Kernel Image (GKI) architecture, this version must be used for the primary boot image.
Android 10 must use boot header version 2.
Android 9 must use boot header version 1.
Android 8 and lower are considered as using a boot image header version 0.
For all devices running with Android 9 or higher, the Vendor Test Suite (VTS) checks the format of the boot/recovery image to ensure that the boot image header uses the correct version. To view AOSP details on all supported boot and vendor boot image headers, refer to system/tools/mkbootimg/include/bootimg/bootimg.h.

Implement boot image header versioning
The mkbootimg tool accepts the following arguments.

Argument Description
header_version Sets the boot image header version. A boot image with a header version:
1 or 2 supports a recovery DTBO image or a recovery ACPIO image.
3 doesn't support recovery images.
recovery_dtbo Used for architectures that use DTB. Specifies the path to the recovery DTBO image. Optional for A/B devices, which don't need a recovery image. Non-A/B devices using header_version:
1 or 2 can specify this path or use the recovery_acpio section to specify a path to a recovery ACPIO image.
3 can't specify a recovery DTBO image.
recovery_acpio Used for architectures that use ACPI instead of DTB. Specifies the path to the recovery ACPIO image. Optional for A/B devices, which don't need a recovery image. Non-A/B devices using header_version:
1 or 2 can specify this path or use the recovery_dtbo section to specify a path to a recovery DTBO image.
3 can't specify a recovery ACPIO image.
dtb Path to the DTB image that's included in the boot/recovery images.
dtb_offset When added to the base argument, provides the physical load address for the final device tree. For example, if the base argument is 0x10000000 and the dtb_offset argument is 0x01000000, the dtb_addr_field in the boot image header is populated as 0x11000000.
The device BoardConfig.mk uses the config BOARD_MKBOOTIMG_ARGS to add header version to the other board-specific arguments of mkbootimg. For example:

BOARD_MKBOOTIMG_ARGS := --ramdisk_offset $(BOARD_RAMDISK_OFFSET) --tags_offset $(BOARD_KERNEL_TAGS_OFFSET) --header_version $(BOARD_BOOTIMG_HEADER_VERSION)
The Android build system uses the BoardConfig variable BOARD_PREBUILT_DTBOIMAGE to set the argument recovery_dtbo of the mkbootimg tool during the creation of the recovery image. For details on the Android Open Source Project (AOSP) changes, review the associated changelists for boot image header versioning.

Boot image header, version 4
Android 12 provides a boot_signature in the boot image header version 4, which can be used to check the integrity of the kernel and the ramdisk. The check is done in VtsSecurityAvbTest and is required for devices using the GKI architecture. However, the boot_signature isn't involved in the device-specific verified boot process and is only used in VTS. See GKI boot.img board configuration and GKI verified boot settings for details.

Vendor boot image header version 4 supports multiple vendor ramdisk fragments.

Version 4 of the boot image header version uses the following format.

```
struct boot_img_hdr
{
#define BOOT_MAGIC_SIZE 8
    uint8_t magic[BOOT_MAGIC_SIZE];
```

```
    uint32_t kernel_size;    /* size in bytes */
    uint32_t ramdisk_size;   /* size in bytes */

    uint32_t os_version;

    uint32_t header_size;    /* size of boot image header in bytes */
    uint32_t reserved[4];
    uint32_t header_version; /* offset remains constant for version check */

#define BOOT_ARGS_SIZE 512
#define BOOT_EXTRA_ARGS_SIZE 1024
    uint8_t cmdline[BOOT_ARGS_SIZE + BOOT_EXTRA_ARGS_SIZE];

    uint32_t signature_size; /* size in bytes */
};
```

Boot image header, version 3
Android 11 updates the boot image header to version 3, which removes the following data:

Second-stage bootloader. The second_size and second_addr fields no longer appear in the boot image header. Devices with a second-stage bootloader must store that bootloader in its own partition.

Recovery image. The requirement for specifying a recovery image has been deprecated, and the recovery_dtbo_size, recovery_dtbo_offset, recovery_acpio_size, and recovery_acpio_offset fields no longer appear in the boot image header.

A/B devices use an update and recovery scheme that makes it unnecessary to specify a DTBO or ACPIO image for recovery.

Non-A/B devices that want to specify a recovery image (either DTBO or ACPIO) should use boot image header version 1 or 2.

Device tree blob (DTB). The DTB is stored in the vendor boot partition, so the dtb_size and dtb_addr fields no longer appear in the boot image header (but are present in the vendor boot image header).

Devices can use boot image header version 3 to comply with Generic Kernel Image (GKI) architecture, which unifies the core kernel and moves vendor modules that are required for boot to the vendor_boot partition (meaning the boot image contains only GKI components). Devices that:

Use GKI (requires the android-4.19 or android-5.4 kernel) but don't use A/B updates can specify a recovery image by using boot image version 3 for the boot image and boot image version 2 for the recovery image.

Don't use GKI and don't use A/B updates can specify a recovery image by using boot image version 1 or 2 for both boot and recovery images.

Version 3 of the boot image header version uses the following format.

```
struct boot_img_hdr
{
#define BOOT_MAGIC_SIZE 8
    uint8_t magic[BOOT_MAGIC_SIZE];

    uint32_t kernel_size;    /* size in bytes */
    uint32_t ramdisk_size;   /* size in bytes */

    uint32_t os_version;

    uint32_t header_size;    /* size of boot image header in bytes */
    uint32_t reserved[4];
    uint32_t header_version; /* offset remains constant for version check */

#define BOOT_ARGS_SIZE 512
#define BOOT_EXTRA_ARGS_SIZE 1024
    uint8_t cmdline[BOOT_ARGS_SIZE + BOOT_EXTRA_ARGS_SIZE];
};
```

Boot image header, version 2
Android 10 updates the boot image header to version 2, which adds a section for recovery DTB image information (image size and physical load address).

Version 2 of the boot image header version uses the following format.

```
struct boot_img_hdr
{
    uint8_t magic[BOOT_MAGIC_SIZE];
    uint32_t kernel_size;            /* size in bytes */
    uint32_t kernel_addr;            /* physical load addr */

    uint32_t ramdisk_size;           /* size in bytes */
    uint32_t ramdisk_addr;           /* physical load addr */

    uint32_t second_size;            /* size in bytes */
    uint32_t second_addr;            /* physical load addr */

    uint32_t tags_addr;              /* physical addr for kernel tags */
    uint32_t page_size;              /* flash page size we assume */
    uint32_t header_version;
    uint32_t os_version;
    uint8_t name[BOOT_NAME_SIZE];    /* asciiz product name */
    uint8_t cmdline[BOOT_ARGS_SIZE];
    uint32_t id[8];                  /* timestamp / checksum / sha1 / etc */
    uint8_t extra_cmdline[BOOT_EXTRA_ARGS_SIZE];
    uint32_t recovery_[dtbo|acpio]_size;    /* size of recovery image */
    uint64_t recovery_[dtbo|acpio]_offset;  /* offset in boot image */
    uint32_t header_size;            /* size of boot image header in bytes */
    uint32_t dtb_size;               /* size of dtb image */
    uint64_t dtb_addr;               /* physical load address */
};
```

Boot image header, version 1

Android 9 converts the unused field of the boot image header to a header version field. Devices launching with Android 9 must use the boot image header with the header version set to 1 or higher (this is verified by VTS).

Version 1 of the boot image header version uses the following format.

```
struct boot_img_hdr
{
    uint8_t magic[BOOT_MAGIC_SIZE];
    uint32_t kernel_size;            /* size in bytes */
    uint32_t kernel_addr;            /* physical load addr */
    uint32_t ramdisk_size;           /* size in bytes */
    uint32_t ramdisk_addr;           /* physical load addr */

    uint32_t second_size;            /* size in bytes */
    uint32_t second_addr;            /* physical load addr */

    uint32_t tags_addr;              /* physical addr for kernel tags */
    uint32_t page_size;              /* flash page size we assume */
    uint32_t header_version;
    uint32_t os_version;
    uint8_t name[BOOT_NAME_SIZE];    /* asciiz product name */
    uint8_t cmdline[BOOT_ARGS_SIZE];
    uint32_t id[8];                  /* timestamp / checksum / sha1 / etc */
    uint8_t extra_cmdline[BOOT_EXTRA_ARGS_SIZE];
    uint32_t recovery_[dtbo|acpio]_size;    /* size of recovery image */
    uint64_t recovery_[dtbo|acpio]_offset;  /* offset in boot image */
    uint32_t header_size;            /* size of boot image header in bytes */
};
```

Non-A/B devices can specify a DTB/ACPI overlay image for recovery to help mitigate over-the-air (OTA) update failures. (A/B devices don't have this problem and don't need to specify an overlay image.) You can specify either a DTBO image or an ACPIO image, but not both (because they're used by different architectures). To configure the boot image header correctly, when using:

A DTBO image for recovery, include the recovery_dtbo_size and recovery_dtbo_offset fields (and don't include the recovery_acpio_size and recovery_acpio_offset fields).

An ACPIO image for recovery, include the recovery_acpio_size and recovery_acpio_offset fields (and don't include the recovery_dtbo_size and recovery_dtbo_offset fields).

The header_size field contains the size of the boot image header. If the boot image header version is set to 1, the id field contains the SHA-1 digest for the recovery_[dtbo|acpio] section of the boot image in addition to the kernel, ramdisk, and second sections. For details on the recovery_[dtbo|acpio]_size and recovery_[dtbo|acpio]_offset fields, see Recovery Images.

Legacy boot image header, version 0
Devices launched before Android 9 using the legacy boot image header are considered as using a boot image header version 0.

```
struct boot_img_hdr
{
    uint8_t magic[BOOT_MAGIC_SIZE];
    uint32_t kernel_size;              /* size in bytes */
    uint32_t kernel_addr;              /* physical load addr */

    uint32_t ramdisk_size;             /* size in bytes */
    uint32_t ramdisk_addr;             /* physical load addr */

    uint32_t second_size;              /* size in bytes */
    uint32_t second_addr;              /* physical load addr */

    uint32_t tags_addr;                /* physical addr for kernel tags */
    uint32_t page_size;                /* flash page size we assume */
    uint32_t unused;
    uint32_t os_version;
    uint8_t name[BOOT_NAME_SIZE];      /* asciiz product name */
    uint8_t cmdline[BOOT_ARGS_SIZE];
    uint32_t id[8];                    /* timestamp / checksum / sha1 / etc */
    uint8_t extra_cmdline[BOOT_EXTRA_ARGS_SIZE];
};
```

Let’s explain the **boot image header** in **simple words** — what it is, why it exists, and how it has evolved across Android versions (from version 0 to 4), using **examples and comparison tables**.

🧠 What is a Boot Image Header?

When Android boots, it loads the **boot image** which contains:

- The **Linux kernel**
- The **ramdisk** (a small filesystem used during early boot)
- Sometimes **device tree** or **recovery data**

But before loading anything, the bootloader needs to understand **what's inside** the boot image and **where** things are.

That’s what the **boot image header** is for:
🧭 It's a **map or instruction sheet** at the start of the boot image that tells the bootloader:

- How big each section is (kernel, ramdisk, etc.)
- Where each section is placed
- What version the boot image format follows

🚦 Why Versioning Is Needed?

As Android evolved, the **structure of the boot image** changed:

- New features were added
- Some fields were removed
- **GKI (Generic Kernel Image)** was introduced

So Android introduced **boot image header versions** to keep track of changes and remain **backward compatible**.

📅 Android Boot Header Version by Android Version

Android Version	Boot Header Version	Key Features
Android 13	3 or 4	GKI devices use **version 4**
Android 12	3 or 4	GKI support via v4
Android 11	3	First GKI-compliant version
Android 10	2	Adds recovery DTBO/ACPIO support
Android 9	1	Adds header_version field
Android 8 or less	0	Original legacy format

🔢 What's Inside the Header? (Simplified)

Field	What it means
`magic`	Identifies this as a boot image
`kernel_size`	Size of the Linux kernel
`ramdisk_size`	Size of the ramdisk
`os_version`	Android OS version and security patch level
`header_version`	Version of this header
`cmdline`	Parameters passed to kernel
`recovery_dtbo_size`, **`recovery_dtbo_offset`**	Used only in v1/v2
`signature_size`	Only in v4, for VTS testing

📦 Example: Version Differences

Let's compare key differences across versions:

Field	v0	v1	v2	v3	v4
`header_version`	❌	✅	✅	✅	✅
`second_stage_loader`	✅	✅	✅	❌	❌
`recovery_dtbo/acpio`	❌	✅	✅	❌	❌
`dtb` in boot image	✅	✅	✅	❌ (moved to vendor boot)	❌
`boot_signature`	❌	❌	❌	❌	✅
Used in GKI	❌	❌	❌	✅	✅

📘 Sample Boot Image Header Code (v3/v4)

```c
struct boot_img_hdr {
    uint8_t  magic[8];            // BOOT_MAGIC
    uint32_t kernel_size;         // in bytes
    uint32_t ramdisk_size;        // in bytes
    uint32_t os_version;          // Android version
    uint32_t header_size;         // size of this header
    uint32_t reserved[4];         // unused/reserved
    uint32_t header_version;      // 3 or 4
    uint8_t  cmdline[1536];       // boot command line
    uint32_t signature_size;      // (only in v4)
};
```

⚙️ Building Boot Images with mkbootimg

When building boot images, the tool **`mkbootimg`** lets you set header version and related options:

```makefile
BOARD_MKBOOTIMG_ARGS := \
  --header_version $(BOARD_BOOTIMG_HEADER_VERSION) \
```

```
  --ramdisk_offset $(BOARD_RAMDISK_OFFSET)
```

📁 Where to See Header in AOSP

- AOSP source: **`system/tools/mkbootimg/include/bootimg/bootimg.h`**
- GKI structure: **`boot.img`** contains kernel + ramdisk, while vendor-specific stuff is in **`vendor_boot.img`**

✅ Summary in Plain Words

Concept	Explained Simply
Boot image header	A **roadmap** at the start of boot.img telling the bootloader what's inside
Versioning	Needed to support new Android features like GKI
GKI	Android's way to make kernel updates easier and unified
mkbootimg	Tool that creates boot.img with the correct header version
v3/v4	Used in modern Android (11+), especially for GKI devices
v0–v2	Used in older/non-GKI devices, with more legacy fields

Want to see:

- How to read the boot header of a real boot.img?
- How to parse boot image using **`unpack_bootimg.py`**?
- How GKI splits boot.img and vendor_boot.img?

Let me know!

6. Implement bootconfig in Android 12

bookmark_border
In Android 12, the bootconfig feature replaces the androidboot.* kernel cmdline options in use with Android 11 and lower. The bootconfig feature is a mechanism for passing configuration details from the build and bootloader to Android 12.

This feature provides a way to separate the configuration parameters for Android user-space from those for the kernel. Moving the lengthy androidboot.* kernel parameters to the bootconfig file creates space on the kernel cmdline and makes it available for future expansion.

Both the kernel and the Android user-space must support bootconfig.

First release that has this support: Android 12
First kernel version that has this support: 12-5.4.xx kernel
Implement the bootconfig feature for new devices launching with a 12-5.10.xx kernel version. You don't need to implement it if you're upgrading devices.

Examples and source
As you view the examples and source code in this section, note that the format of the bootconfig code is only slightly different from the format of the kernel cmdline used in Android 11 and lower. However, the following difference is important for your usage:

Parameters must be separated by the newline escape sequence \n, not by spaces.
Bootloader example
For a bootloader example, see the Cuttlefish U-boot reference bootloader implementation. Two commits in the reference are listed below. The first uprevs the boot header version support to the latest version. In the example, the first commit updates (or uprevs)the version support to the next one, v4. The second does two things; it adds bootconfig handling, and demonstrates adding parameters at runtime:

Uprev the boot header version support to the latest v4 version.
Add the bootconfig handling.
Build example
For a build example that shows mkbootimg changes to build the vendor_boot.img with vendor boot header v4, see mkbootimg changes for bootconfig. See the Cuttlefish changes to do the following:

Use (or uprev to) the vendor boot header version v4.
Add bootconfig to the kernel cmdline and move selected parameters to bootconfig.
Implementation
Partners must add support to their bootloaders, and move their build-time androidboot.* parameters from the kernel cmdline to the bootconfig file. The best way to implement this change is to do so incrementally; see the Incremental implementation and validation section for information on following an incremental process.

If you have changes that search the /proc/cmdline file for androidboot.* parameters, point them to the /proc/bootconfig file instead. The ro.boot.* properties are set with the new bootconfig values, so you don't need to make changes for code using those properties.

Build changes
First, uprev your boot header version to version 4:

- BOARD_BOOT_HEADER_VERSION := 3

+ BOARD_BOOT_HEADER_VERSION := 4

Add the bootconfig kernel cmdline parameter. This makes the kernel look for the bootconfig section:

BOARD_KERNEL_CMDLINE += bootconfig

The bootconfig parameters are created from the parameters in the BOARD_BOOTCONFIG variable, much like the kernel cmdline is created from BOARD_KERNEL_CMDLINE.

Any androidboot.* parameters can be moved as-is, similar to the following:

- BOARD_KERNEL_CMDLINE += androidboot..selinux=enforcing

+ BOARD_BOOTCONFIG += androidboot..selinux=enforcing

Bootloader changes

The bootloader sets up the initramfs before jumping to the kernel. The kernel boot configuration searches for the bootconfig section, and looks for it to be at the very end of initramfs, with the expected trailer.

The bootloader gets the vendor_boot.img layout information from the vendor boot image header.

Diagram of bootconfig memory allocation layout

Figure 1. Android 12 bootconfig memory allocation

The bootloader creates the bootconfig section in memory. The bootconfig section contains memory allocations for the following:

Parameters
4 B size parameters size
4 B size parameters checksum
12 B bootconfig magic string (#BOOTCONFIG\n)
The parameters come from two sources: Parameters known at build time, and parameters that aren't known at build time. Unknown parameters must be added.

Parameters known at build time are packaged into the end of the vendor_boot image in the bootconfig section. The size of the section is stored (as bytes) in the vendor boot header field vendor_bootconfig_size.

The parameters that aren't known at build time are only known at runtime in the bootloader. These must be added to the end of the bootconfig parameters section before the bootconfig trailer is applied.

If you need to add any parameters after the bootconfig trailer has been applied, overwrite the trailer and reapply it.

Incremental implementation and validation
Implement the bootconfig feature incrementally by following the process given in this section. Leave the kernel cmdline parameters untouched while the bootconfig parameters are added.

Important: If the bootconfig function fails to load, and the androidboot.* parameters aren't in the kernel cmdline, then the device might not boot. Many of these parameters are very important to Android boot. Therefore, Google recommends that you implement this feature incrementally to make sure the bootloader and build changes work before removing the kernel cmdline parameters.
These are the steps for an incremental implementation, with validation:

Make the bootloader and build changes, then do the following:
Use the BOARD_BOOTCONFIG variable to add a new bootconfig parameter.
Keep the kernel cmdline parameters the way they are, so that the device can continue to boot correctly. This makes debugging and validation much easier.
Verify your work by checking the contents of /proc/bootconfig. Verify that you see the newly added parameter after the device has booted.
Move the androidboot.* parameters from the kernel cmdline to bootconfig, using the BOARD_BOOTCONFIG variable and the bootloader.
Verify that each of the parameters exists in /proc/bootconfig AND that they aren't in /proc/cmdline. If you can verify this, your implementation was successful.
OTA upgrading and downgrading considerations
When you're managing OTA upgrades and downgrades between different versions of Android, or different kernel versions, special care should be taken.

Android 12 is the first version with bootconfig support. If downgrading to any version before that, kernel cmdline parameters must be used instead of bootconfig.

Kernel versions 12-5.4 and later support bootconfig. If downgrading to any version before that(including 11-5.4), kernel cmdline parameters must be used.

Upgrades from Android 11 and lower to Android 12 and higher can keep using kernel cmdline parameters. The same goes for upgrading kernel versions.

Troubleshooting
When you perform the verify step, if you don't see the expected parameters in /proc/bootconfig, check the kernel logs in logcat. There's always a log entry present for bootconfig if the kernel supports it.

Example log output

$ adb logcat | grep bootconfig
02-24 17:00:07.610   0   0 I Load bootconfig: 128 bytes 9 nodes
If you see an error log returned, then there was a problem loading the bootconfig. To see different error types, view init/main.c.

Here's a **simple explanation** of how to **implement bootconfig in Android 12** for a new device, what it does, why it's useful, and step-by-step guidance with examples.

🧠 What is Bootconfig?

Bootconfig is a **new way of passing boot-time parameters** to Android in **Android 12 and kernel 5.10+**.

Before Android 12:

- Boot parameters (like `androidboot.selinux=enforcing`) were passed through **kernel command line** (in `/proc/cmdline`)
- This had length limits and mixed kernel + Android settings

In Android 12+:

- `androidboot.*` parameters are now stored in a **separate section** called **bootconfig**, accessible via `/proc/bootconfig`

✅ Why Use Bootconfig?

Problem with old method	Solution with bootconfig
Kernel command line too long	Frees up space by separating Android config
Mixed kernel and Android args	Keeps Android args in one place
Hard to manage	Easier to parse, organize, and validate

🛠️ What You Need To Do (Summary)

1. 🔼 Upgrade boot image header to version 4

In `BoardConfig.mk`:

```make
BOARD_BOOT_HEADER_VERSION := 4
```

2. ➕ Enable bootconfig support in the kernel

Also in `BoardConfig.mk`:

```make
BOARD_KERNEL_CMDLINE += bootconfig
```

3. 📦 Move androidboot.* params to bootconfig

Before:

```make
BOARD_KERNEL_CMDLINE += androidboot.selinux=enforcing
```

After:

```make
BOARD_BOOTCONFIG += androidboot.selinux=enforcing
```

This tells the build system to pack these into the **bootconfig section** of the `vendor_boot.img`.

👟 Bootloader Changes

The **bootloader** must:

- Detect that the device uses boot header version 4
- Read **`vendor_bootconfig_size`** from **`vendor_boot.img`**
- Copy the bootconfig section into memory
- Append any dynamic parameters (like slot info)
- Add the bootconfig trailer:

  ```python
  4 bytes: size
  4 bytes: checksum
  12 bytes: #BOOTCONFIG\n
  ```

📌 Example bootconfig memory layout:

```css
[ bootconfig parameters ]
[    size (4B)         ]
[  checksum (4B)       ]
[ "#BOOTCONFIG\n" (12B)]
```

This section is placed at the **end of the initramfs** so the kernel can find it during boot.

🔄 Incremental Implementation Strategy

You should **migrate gradually** to bootconfig to avoid boot failures:

Step-by-step:

1. Keep old kernel command line parameters (device boots as usual)
2. Add 1 new bootconfig parameter using **`BOARD_BOOTCONFIG += foo.bar=test`**
3. Rebuild and boot device
4. Run:

   ```bash
   adb shell cat /proc/bootconfig
   ```

   ✔️ Check that **`foo.bar=test`** appears
   ✔️ Also check **`/proc/cmdline`** – it **should not** have it
5. Now *move androidboot. parameters** to **`BOARD_BOOTCONFIG`**
6. Ensure:
   - **`/proc/cmdline`** does NOT have them
   - **`/proc/bootconfig`** DOES

🎯 If all is good, remove those parameters from the cmdline.

🧪 Verification

Check bootconfig loaded:

```bash
adb logcat | grep bootconfig
```

You should see something like:

```css
I Load bootconfig: 128 bytes 9 nodes
```

If not, check the kernel logs for parsing errors.

🧯 Downgrade/OTA Notes

Condition	What to use
Android < 12	kernel cmdline only
Kernel < 5.4	kernel cmdline only
Upgrading from Android 11 → 12	You can keep using cmdline, or migrate to bootconfig gradually
Downgrading to Android 11	You **must** move parameters back to cmdline

🧱 Example `BoardConfig.mk`

```make
BOARD_BOOT_HEADER_VERSION := 4
BOARD_KERNEL_CMDLINE += bootconfig console=ttyMSM0
BOARD_BOOTCONFIG += \
  androidboot.verifiedbootstate=green \
  androidboot.slot_suffix=_a \
  androidboot.hardware=qcom \
  androidboot.selinux=enforcing
```

📂 Example /proc/bootconfig Output

```text
androidboot.verifiedbootstate=green
androidboot.slot_suffix=_a
androidboot.hardware=qcom
androidboot.selinux=enforcing
```

✅ Summary Table

Task	Tool/Location	Example
Enable bootconfig	`BoardConfig.mk`	`BOARD_KERNEL_CMDLINE += bootconfig`
Add bootconfig params	`BOARD_BOOTCONFIG`	`androidboot.selinux=enforcing`
Upgrade header	`BoardConfig.mk`	`BOARD_BOOT_HEADER_VERSION := 4`
Verify success	ADB	`adb shell cat /proc/bootconfig`
Fallback	Use both temporarily	Kernel cmdline + bootconfig

Let me know if you want:

- A full example bootloader patch
- How to generate boot.img and vendor_boot.img with `mkbootimg`
- C code to create bootconfig trailer with size and checksum

7. ↪
androidboot.slot_suffix=_a
why we use this

We use **`androidboot.slot_suffix=_a`** to **tell Android which boot slot (A or B)** the device is currently booting from. It is part of the **A/B (Seamless) Update System** introduced in Android to allow updates without downtime or risk of bricking.

🧠 What is `slot_suffix`?

In devices that support **A/B partitioning**, each critical partition exists in two copies:

- **`boot_a`** and **`boot_b`**
- **`system_a`** and **`system_b`**
- **`vendor_a`** and **`vendor_b`**
- and so on...

This allows the system to:

- Install updates to the **inactive** slot (say, **`_b`**)
- Reboot into it
- If it works, keep it
- If it fails, **rollback** to the previous slot (**`_a`**)

🧾 Why `androidboot.slot_suffix=_a`?

- It is **set by the bootloader**
- Passed to the kernel via **cmdline** or **bootconfig**
- Android reads it and knows:
  - Which slot to mount
  - Where to find the correct partitions (**`system_a`**, **`boot_a`**, etc.)
  - How to set **`ro.boot.slot_suffix`** and **`ro.boot.slot`** system properties

✅ Example

If **`androidboot.slot_suffix=_b`** is passed, Android will:

- Mount **`system_b`**
- Use **`boot_b`**, **`vendor_b`**, etc.
- Set:

  ```bash
  getprop ro.boot.slot_suffix  → _b
  getprop ro.boot.slot         → b
  ```

🔁 During OTA Updates

The flow:

1. You boot from **`_a`**
2. OTA updates **`_b`**
3. Sets **`_b`** as active
4. On next boot, **`androidboot.slot_suffix=_b`** is passed

5. Android boots from `_b`
6. If boot succeeds → stays on `_b`
7. If boot fails → reverts to `_a`

🧪 How to Check on a Running Device

```bash
adb shell getprop ro.boot.slot_suffix
```

🧱 Summary

Key	Description
`androidboot.slot_suffix`	Tells Android which slot (**_a** or **_b**) is active
Who sets it	Bootloader
Where used	Kernel cmdline or bootconfig
Why important	Enables seamless A/B updates and rollback

Let me know if you want to simulate A/B behavior or inspect OTA flows.

8. DTB images

bookmark_border
Android implementations can include a device tree blob (DTB) image for use by the bootloader. The location of the DTB image (and the options for specifying DTB image parameters) differs between Android releases.

In Android 11, devices using the Generic Kernel Image (GKI) must support the vendor boot partition, which includes all vendor-specific information that was relocated from the boot partition. Because the DTB image contains vendor-specific data, it's now part of the vendor boot partition. To specify DTB image parameters, see Vendor boot header.

In Android 10, devices can include the DTB image in the boot partition. To specify DTB image parameters, see Including the DTB image in the boot image.

In Android 9 and lower, the DTB image can exist in its own partition or be appended to the kernel image.gz to create the kernel + DTB image (which is then passed to mkbootimg to create boot.img).

DTB image format
In Android 10 and higher, the DTB image must use one of the following formats:

DT blobs concatenated one after the other. The bootloader uses the totalsize field in each FDT header to read and parse the corresponding blob.

DTB/DTBO partitions. The bootloader has an efficient way to select the correct DT blob by examining the dt_table_entry struct (contains id, rev, and custom fields) that can hold hardware identifying information for the entry. For details, see DTB/DTBO Partitions.

Include the DTB image in the boot image
Devices running Android 10 can include the DTB image in the boot image. This removes the need for Android to support scripts that append the DTB image to image.gz in the kernel, and enables the use of Vendor Test Suite (VTS) test to verify (and standardize) DTB placement.

In addition, for non-A/B devices, it's safer to have the DTB as part of the recovery image rather than in a separate partition to prevent issues caused by interrupted OTAs. During an OTA, if a problem occurs after the DTB partition is updated (but prior to completing the full update), the device tries to boot into recovery mode to complete the OTA; however, because the DTB partition has already been updated, a mismatch could occur with the recovery image (which has not yet been updated). Having the DTB image as part of the boot image format prevents such issues by making the recovery image self sufficient (that is, it doesn't depend on another partition).

Boot image structure
Devices running Android 10 can include a DTB image using the following boot image structure.

Boot image section	Number of pages
Boot header (1 page)	1
Kernel (l pages)	l = (kernel_size + page_size - 1)/page_size
Ramdisk (m pages)	m = (ramdisk_size + page_size - 1)/page_size
Second stage bootloader (n pages)	n = (second_size + page_size - 1)/page_size
Recovery DTBO (o pages)	o = (recovery_dtbo_size + page_size - 1)/page_size
DTB (p pages)	p = (dtb_size + page_size - 1)/page_size

DTB image path
For devices running Android 10, you can use the mkbootimg.py tool and the following arguments to specify the path to the DTB image.

Argument	Description
dtb	Path to the DTB image to be included in the boot/recovery images.
dtb_offset	When added to the base argument, provides the physical load address for the final device tree. For example, if the base argument is 0x10000000 and the dtb_offset argument is 0x01000000, the dtb_addr_field in the boot image header is populated as 0x11000000.

The board config variable BOARD_PREBUILT_DTBIMAGE_DIR must be used to specify the path to the DTB image. If more than one file with extension *.dtb is present in the directory BOARD_PREBUILT_DTBIMAGE_DIR, the Android build system concatenates the files to create the final DTB image used in the boot image creation.

To pass the argument dtb to mkbootimg.py with the DTB image from the directory specified by BOARD_PREBUILT_DTBIMAGE_DIR, the board config variable BOARD_INCLUDE_DTB_IN_BOOTIMG must be set to true. For example:

BOARD_INCLUDE_DTB_IN_BOOTIMG := true
You can append the dtb_offset argument to the BOARD_MKBOOTIMG_ARGS board config variable with the other offsets and header version. For example:

BOARD_MKBOOTIMG_ARGS := --ramdisk_offset $(BOARD_RAMDISK_OFFSET) --dtb_offset $(BOARD_DTB_OFFSET) --tags_offset $(BOARD_KERNEL_TAGS_OFFSET) --header_version $(BOARD_BOOTIMG_HEADER_VERSION)
Bootloader support
For VTS to run successfully on devices running Android 10, the bootloader must support the updated boot image and must add the androidboot.dtb_idx kernel command line parameter to indicate the index of the selected device tree (DT). You can specify only one (1) index. For example, the parameter androidboot.dtb_idx=N reports N as the zero-based index of the device tree selected by the bootloader from the set of DTBs present in the boot image.

🧠 Why Do We Use a DTB (Device Tree Blob) Image in Android?

In simple words:

A **DTB (Device Tree Blob)** tells the **Linux kernel** what kind of hardware the device has — like the CPU, memory, I/O buses, display, etc. This allows the **same kernel binary** to run on **different hardware**, by just giving it the right "map" (DTB) of the system.

🚀 What Problem Does DTB Solve?

Before DTB, hardware information was hardcoded into the kernel. That made:

- Kernel updates difficult
- Code messy and hardware-dependent

DTB separates the hardware description from the kernel code. Now, the kernel is more generic, and the device tree (DTB) provides device-specific info at boot time.

📦 Where Is DTB Stored in Android?

It depends on the **Android version**:

Android Version	DTB Location
≤ Android 9	- Appended to **`kernel.gz`** - Or in a separate **`dtb`** partition
Android 10	- Inside the **`boot.img`** (via **`mkbootimg --dtb`**)
Android 11+ (GKI)	- Inside **`vendor_boot.img`** (moved from **`boot.img`**)

📑 What's in a DTB?

A DTB contains:

- CPU type
- Memory layout
- GPIOs
- I2C/SPI devices
- Display info
- ...any hardware-specific info

🛠️ How Android Builds and Uses DTB

For Android 10 Devices:

1. Build system collects all **`*.dtb`** files.
2. Concatenates them → final **`dtb.img`**
3. Adds it to the **`boot.img`** using:

```make
BOARD_INCLUDE_DTB_IN_BOOTIMG := true
BOARD_PREBUILT_DTBIMAGE_DIR := path/to/dtb/files
BOARD_MKBOOTIMG_ARGS += --dtb_offset $(BOARD_DTB_OFFSET)
```

4. **`mkbootimg`** includes the DTB at the correct offset.

Example Boot Image Layout:

Section	What it contains
Boot Header	Metadata
Kernel	Linux kernel
Ramdisk	Initramfs
Second Stage Bootloader	Optional stage2 code
Recovery DTBO	DT overlay for recovery
DTB	Device hardware info

🧰 Bootloader Responsibilities

At boot time, the **bootloader must:**

- Choose the correct DTB (e.g., based on hardware ID)
- Pass the DTB to the kernel
- Set the DTB index in cmdline:

  ```bash
  androidboot.dtb_idx=2
  ```

🔍 DTB Formats Supported

1. **Concatenated FDT blobs**
   - One after another, parsed by **`totalsize`** field
2. **DTB/DTBO partitions**
   - With a **`dt_table_entry`** describing hardware IDs

✅ Why It’s Important

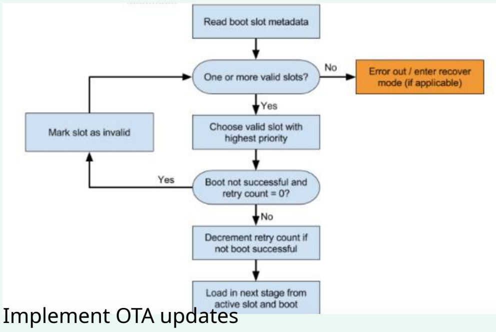
- Makes Android OS more modular
- Enables **Generic Kernel Image (GKI)**: same kernel, different vendors
- Improves OTA update safety
- Simplifies recovery/boot processes

📊 Summary

Term	Meaning
DTB	Device Tree Blob (hardware map)
DTBO	Device Tree Blob Overlay (modifications on top of base DTB)
DT	Human-readable **`.dts`** files compiled into **`.dtb`**
GKI	Generic Kernel Image (Android 11+ modular kernel)
`androidboot.dtb_idx`	Kernel parameter showing selected DTB index

Let me know if you'd like a flow diagram or code samples!

9.



Implement OTA updates

bookmark_border
To implement the over-the-air (OTA) updates, the bootloader must be able to access a recovery RAM disk during boot. If the device uses an unmodified AOSP recovery image, the bootloader reads the first 32 bytes on the misc partition; if the data there matches boot-recovery, the bootloader boots into the recovery image. This method enables any pending recovery work (for example, applying an OTA or removing data) to continue to completion.

For details on the content of a block in flash used for communications by recovery and the bootloader, refer to bootable/recovery/bootloader_message/bootloader_message.h.

Devices with A/B updates
To support OTA updates on devices that use A/B updates, ensure that the device bootloader meets the following criteria.

General criteria
All partitions updated through an OTA should be updatable while the main system is booted (and not updated in recovery).

To boot the system partition, the bootloader passes the following value on kernel command line: ro root=/dev/[node] rootwait init=/init.

It's the responsibility of the Android framework to call markBootSuccessful from the HAL. The bootloader should never mark a partition as successfully booted.

Support for boot control HAL
The bootloader must support the boot_control HAL as defined in hardware/libhardware/include/hardware/boot_control.h. The updater queries the boot control HAL, updates the boot slot not in use, changes the active slot using the HAL, and reboots into the updated operating system. For details, see Implementing the boot control HAL.

Support for slots
The bootloader must support functionality related to partitions and slots, including:

Partition names must include a suffix that identifies which partitions belong to a particular slot in the bootloader. For each such partition, there's a corresponding variable has-slot:partition base name with a value of yes. Slots are named alphabetically as a, b, c, etc. corresponding to partitions with the suffix _a, _b, _c, etc. The bootloader should inform the operating system which slot was booted using the command line property androidboot.slot_suffix. This property is set through bootconfig for devices launching with Android 12 or higher.

The slot-retry-count value is reset to a positive value (usually 3), either by the boot control HAL through the setActiveBootSlot callback or through the fastboot set_active command. When modifying a partition that's part of a slot, the bootloader clears "successfully booted" and resets the retry count for the slot.

The bootloader should also determine which slot to load. The figure shows an example decision process.

Bootloader slotting flow
Figure 1. Bootloader slotting flow
Determine which slot to attempt. Don't attempt to load a slot marked slot-unbootable. This slot should be consistent with the values returned by fastboot, and is referred to as the current slot.

If the current slot isn't marked as slot-successful and has a slot-retry-count = 0, mark the current slot as slot-unbootable. Then select a different slot that is not marked unbootable and is marked as slot-successful; this slot is now the selected slot. If no current slot is available, boot to recovery or display a meaningful error message to the user.

Select the appropriate boot.img and include the path to correct system partition on the kernel command line.

Populate the kernel command line slot_suffix parameter.

Boot. If not marked slot-successful, decrement slot-retry-count.

The fastboot utility determines which partition to flash when running any flash commands. For example, running the fastboot flash system system.img command first queries the current-slot variable then concatenates the result to system to generate the name of the partition that should be flashed (system_a, system_b, etc.).

When setting the current slot using the fastboot set_active command or the boot control HAL setActiveBootSlot command, the bootloader should update the current slot, clear slot-unbootable and slot-successful, and reset the retry count (this is the only way to clear slot-unbootable).

Devices without A/B updates
To support OTA updates on devices that don't use A/B updates (see Non-A/B updatable devices), ensure that the device bootloader meets the following criteria.

The recovery partition should contain an image that is capable of reading a system image from some supported partition (cache, userdata) and writing it to the system partition.

The bootloader should support booting directly into recovery mode.

If radio image updates are supported, the recovery partition should also be able to flash the radio. This can be accomplished in one of two ways:

The bootloader flashes the radio. In this case, it should be possible to reboot from the recovery partition back into the bootloader to complete the update.

The recovery image flashes the radio. This functionality can be provided as a binary library or utility.

The diagram you've uploaded (`bootloader_slotting.png`) visually represents the **bootloader slot selection and retry mechanism** for **A/B OTA updates** in Android. Here's an explanation of how the bootloader handles **slot metadata** during the boot process, based on that diagram and the accompanying text:

✅ Bootloader Slotting Flow (Step-by-Step)

1. Read Boot Slot Metadata

- Bootloader reads the **slot metadata** stored in the **misc partition** or **persistent storage**.
- Metadata includes:
  - `retry_count`
  - `successful_boot` flag
  - `bootable` flag

---

2. Check for One or More Valid Slots

- **Valid slot** = slot is **bootable** and not marked **unbootable**.

If none are valid →
➡️ Enter **recovery mode** or **show error** (failsafe).

---

3. Choose Slot with Highest Priority

- Among valid slots, select the one with:
  - `successful_boot = true` OR
  - `retry_count > 0`
  - **highest priority** value

---

4. Try to Boot from Chosen Slot

- Attempt boot.

- If boot is **successful**, Android framework will later call **`markBootSuccessful()`** via HAL.

5. Check Boot Outcome

- If **boot fails**:
  - **And** retry count == 0 → mark the slot as **unbootable**
  - **Else**, retry:
    - Decrement **`retry_count`**
    - Try again next boot

6. If Successful:

- Boot proceeds to normal Android boot stage.
- Android framework handles setting:
  - **`slot_successful = true`**
  - No changes are made by bootloader for successful boot itself.

📘 Important Concepts

Field	Meaning
`retry_count`	Number of remaining attempts to boot from the slot.
`successful_boot`	Set **only after** Android boots completely and calls **`markBootSuccessful()`**
`bootable` / **`unbootable`**	Slot considered safe for boot. Marked **unbootable** if retries fail.
`androidboot.slot_suffix`	Kernel cmdline or bootconfig field to tell Android which slot is booted (**_a**, **_b**, etc.).
`fastboot set_active`	Command to manually switch the active slot.

🔁 Summary of A/B OTA Boot Flow

```
mermaid

graph TD
    A[Bootloader reads slot metadata] --> B{Valid slot(s)?}
    B -- No --> E[Enter recovery or error out]
    B -- Yes --> C[Choose highest-priority valid slot]
    C --> D{Boot fails & retry_count = 0?}
    D -- Yes --> F[Mark slot unbootable] --> B
    D -- No --> G[Decrement retry_count]
    G --> H[Load next stage & boot]
```

If you'd like, I can generate a **table of slot metadata fields**, or walk you through how this is implemented in the **boot_control HAL** or the **bootloader code (LK/U-Boot)**. Let me know!

10. Implement OTA updates

bookmark_border
To implement the over-the-air (OTA) updates, the bootloader must be able to access a recovery RAM disk during boot. If the device uses an unmodified AOSP recovery image, the bootloader reads the first 32 bytes on the misc partition; if the data there matches boot-recovery, the bootloader boots into the recovery image. This method enables any pending recovery work (for example, applying an OTA or removing data) to continue to completion.

For details on the content of a block in flash used for communications by recovery and the bootloader, refer to bootable/recovery/bootloader_message/bootloader_message.h.

Devices with A/B updates
To support OTA updates on devices that use A/B updates, ensure that the device bootloader meets the following criteria.

General criteria
All partitions updated through an OTA should be updatable while the main system is booted (and not updated in recovery).

To boot the system partition, the bootloader passes the following value on kernel command line: ro root=/dev/[node] rootwait init=/init.

It's the responsibility of the Android framework to call markBootSuccessful from the HAL. The bootloader should never mark a partition as successfully booted.

Support for boot control HAL
The bootloader must support the boot_control HAL as defined in hardware/libhardware/include/hardware/boot_control.h. The updater queries the boot control HAL, updates the boot slot not in use, changes the active slot using the HAL, and reboots into the updated operating system. For details, see Implementing the boot control HAL.

Support for slots
The bootloader must support functionality related to partitions and slots, including:

Partition names must include a suffix that identifies which partitions belong to a particular slot in the bootloader. For each such partition, there's a corresponding variable has-slot:partition base name with a value of yes. Slots are named alphabetically as a, b, c, etc. corresponding to partitions with the suffix _a, _b, _c, etc. The bootloader should inform the operating system which slot was booted using the command line property androidboot.slot_suffix. This property is set through bootconfig for devices launching with Android 12 or higher.

The slot-retry-count value is reset to a positive value (usually 3), either by the boot control HAL through the setActiveBootSlot callback or through the fastboot set_active command. When modifying a partition that's part of a slot, the bootloader clears "successfully booted" and resets the retry count for the slot.

The bootloader should also determine which slot to load. The figure shows an example decision process.

Bootloader slotting flow
Figure 1. Bootloader slotting flow
Determine which slot to attempt. Don't attempt to load a slot marked slot-unbootable. This slot should be consistent with the values returned by fastboot, and is referred to as the current slot.

If the current slot isn't marked as slot-successful and has a slot-retry-count = 0, mark the current slot as slot-unbootable. Then select a different slot that is not marked unbootable and is marked as slot-successful; this slot is now the selected slot. If no current slot is available, boot to recovery or display a meaningful error message to the user.

Select the appropriate boot.img and include the path to correct system partition on the kernel command line.

Populate the kernel command line slot_suffix parameter.

Boot. If not marked slot-successful, decrement slot-retry-count.

The fastboot utility determines which partition to flash when running any flash commands. For example, running the fastboot flash system system.img command first queries the current-slot variable then concatenates the result to system to generate the name of the partition that should be flashed (system_a, system_b, etc.).

When setting the current slot using the fastboot set_active command or the boot control HAL setActiveBootSlot command, the bootloader should update the current slot, clear slot-unbootable and slot-successful, and reset the retry count (this is the only way to clear slot-unbootable).

Devices without A/B updates
To support OTA updates on devices that don't use A/B updates (see Non-A/B updatable devices), ensure that the device bootloader meets the following criteria.

The recovery partition should contain an image that is capable of reading a system image from some supported partition (cache, userdata) and writing it to the system partition.

The bootloader should support booting directly into recovery mode.

If radio image updates are supported, the recovery partition should also be able to flash the radio. This can be accomplished in one of two ways:

The bootloader flashes the radio. In this case, it should be possible to reboot from the recovery partition back into the bootloader to complete the update.

The recovery image flashes the radio. This functionality can be provided as a binary library or utility.

Implementing **OTA (Over-The-Air) updates** in Android requires different approaches depending on whether the device supports **A/B (Seamless)** or **non-A/B** updates.

Below is a **complete guide** to implementing OTA updates in both cases:

✅ Overview

Type	A/B OTA Devices	Non-A/B OTA Devices
Partitions	system_a / system_b	system, recovery
OTA update location	While system is running	From recovery
Recovery role	Not used	Required
Rollback safety	High (fallback slot)	Low (overwrite active slot)
Bootloader requirements	Boot control HAL + slot logic	Recovery boot support
How radio is updated	Via recovery or bootloader	Via recovery or bootloader

---

🔁 A/B OTA Update Flow

📌 1. Partition Structure

- All critical partitions (boot, system, vendor, etc.) are duplicated:
  - `boot_a`, `boot_b`, `system_a`, `system_b`, etc.
- Suffix `_a`, `_b` identifies slot.

📌 2. Bootloader Must Support

- Reading slot metadata (retry count, successful flag).
- Selecting correct slot.
- Kernel cmdline or bootconfig must include:

```bash
androidboot.slot_suffix=_a (or _b)
```

📌 3. Boot Control HAL

- Located at: `hardware/libhardware/include/hardware/boot_control.h`
- Allows system to:
  - Query current slot
  - Mark current slot successful
  - Set active slot for next boot

⚠️ Bootloader **must never** mark slot successful — this is done **only by Android framework** via `markBootSuccessful()`.

🧠 Slot Selection Logic (From Your Diagram)

```
mermaid

graph TD
  A[Read slot metadata] --> B{Valid slot exists?}
  B -- No --> G[Error or recovery mode]
  B -- Yes --> C[Choose highest priority valid slot]
  C --> D{Boot fails AND retry_count = 0?}
  D -- Yes --> E[Mark slot as unbootable] --> B
  D -- No --> F[Decrement retry count, try again]
  F --> H[Boot from selected slot]
```

📌 4. OTA Flow with boot_control HAL

- OTA runs in Android.
- Downloads update and writes it to **inactive slot** (e.g., `_b`).
- Calls `setActiveBootSlot(_b)` — resets retry count.
- Reboots.
- Bootloader loads `_b`, but **does not** mark as successful.
- Android boots from `_b`, calls `markBootSuccessful()` when verified.
- Only now is `_b` marked as stable.

📌 5. fastboot Integration

- `fastboot flash system system.img` → flashes to current slot (`system_a` or `system_b`).
- `fastboot set_active a` → sets current slot to `_a`, resets retry count.

⚙️ Non-A/B OTA Update Flow

📌 1. Recovery Partition Required

- Recovery image contains logic to:
  - Flash system/radio/vendor partitions.
  - Read update package from `/cache` or `/data`.

📌 2. Bootloader Requirements

- Must boot into **recovery** when needed.
- Typically reads the **first 32 bytes of `misc` partition**:
  - If string is `boot-recovery`, then boot recovery image instead of system.

```cpp
cpp

// bootloader_message.h defines:
char command[32];  // e.g., "boot-recovery"
```

📌 3. Radio Image Updates

Handled via:

- **Bootloader** (preferred): Flash radio directly, then reboot to recovery.
- **Recovery**: Embedded binary flashes modem firmware from update.zip.

🔐 Failsafe Mechanism

Condition	Bootloader Action
All slots unbootable	Boot to recovery
Retry count == 0	Mark slot unbootable
No successful slot found	Show error screen
OTA applied to bad slot	Fallback to good slot (A/B only)

📁 Key Paths in AOSP

Purpose	Path
Boot Control HAL	`hardware/libhardware/include/hardware/boot_control.h`
bootloader_message struct	`bootable/recovery/bootloader_message/bootloader_message.h`
A/B OTA update logic	`system/update_engine/`
Recovery flow	`bootable/recovery/`
Boot metadata	`system/core/fs_mgr/include/fs_mgr/slotselect.h`

✅ Example Kernel Command Line (A/B device)

```bash
androidboot.slot_suffix=_b
ro root=/dev/mmcblk0p17 rootwait init=/init
```

✅ Summary

Feature	A/B OTA	Non-A/B OTA
Duplicated partitions	✅	❌
Uses recovery for OTA	❌	✅
`boot_control` HAL required	✅	❌
Retry + slot fallback	✅	❌
Update while running	✅	❌
Bootloader parses `misc`	❌	✅

Let me know if you'd like:

- Sample implementation of `boot_control HAL`
- How `update_engine` applies OTA zip
- Flow of `markBootSuccessful` call
- Recovery `bootloader_message` structure explained in code

I'm happy to guide you step-by-step.

11. ↪
⚙️ Non-A/B OTA Update Flow ------------------------- ### 📌 1. **Recovery Partition Required** * Recovery image contains logic to: * Flash system/radio/vendor partitions. * Read update package from `/cache` or `/data`. ### 📌 2. **Bootloader Requirements** * Must boot into **recovery** when needed. * Typically reads the **first 32 bytes of `misc` partition**: * If string is `boot-recovery`, then boot recovery image instead of system. cpp CopyEdit `// bootloader_message.h defines: char command[32]; // e.g., "boot-recovery"` ### 📌 3. **Radio Image Updates** Handled via: * **Bootloader** (preferred): Flash radio directly, then reboot to recovery. * **Recovery**: Embedded binary flashes modem firmware from update.zip.
explain in simple words

Sure! Here's a **simple explanation** of the **non-A/B OTA update process** — the older update method used on many Android devices:

🔁 What is Non-A/B OTA?

This is the traditional way Android updates itself. Instead of having two copies of system partitions (like in A/B), there's **only one set**, and Android uses **recovery mode** to install updates safely.

⚙️ How It Works (Step by Step)

📌 1. Recovery Partition is Required

- Android phones have a special mode called **Recovery**.
- This recovery system can:
  - Read an update file (like an update.zip) from `/cache` or `/data`.
  - Install (flash) that update onto the main Android system.
  - Flash other components too, like radio (modem) firmware.

  Think of recovery like a mini-Android whose job is to install updates.

📌 2. Bootloader's Job

- When the phone powers on, the **bootloader** decides whether to:
  - Boot normal Android, or
  - Boot into recovery mode (to apply an update).

🧠 How does bootloader know?

- There's a tiny "message" area in memory called the **misc partition**.
- Android can write `"boot-recovery"` into this area.
- When bootloader reads it and sees `"boot-recovery"`, it boots into **recovery**, not Android.

```c
// In C code (bootloader_message.h), this looks like:
char command[32];  // could contain "boot-recovery"
```

📌 3. Updating Radio (Modem) Firmware

- Some updates also include new software for the phone's modem (radio).
- This can be flashed in **two ways**:

✅ Option 1: Bootloader flashes the radio

- Bootloader directly writes the new modem firmware.
- Then reboots to recovery to complete the update.

✅ **Option 2: Recovery flashes the radio**

- The recovery image contains a tool or binary to flash the modem firmware from the update package (zip file).

🧠 Why Recovery is Important in Non-A/B?

Because there's only **one copy of the system**, you can't update Android while it's running. So:

- Boot into **recovery mode**.
- Let recovery install the update.
- Reboot back into Android.

✅ Summary (in very simple terms):

Thing	Meaning
Recovery	A special update mode
Bootloader	Chooses whether to start Android or Recovery
Misc partition	Tiny message area to tell bootloader what to do
Update.zip	Contains the new Android version and sometimes modem firmware
Modem update	Done by bootloader or recovery

Would you like a diagram or code example to make it even clearer?

12. ↪
Retry coun
explain in simple words

Sure! Let's explain **retry count** in very simple words.

🔁 What is Retry Count in Android OTA (A/B update system)?

When Android updates using the **A/B slot system** (two system copies: `slot_a` and `slot_b`), it needs to make sure the new system (the one just updated) **boots successfully**.

To handle this safely, Android uses a **retry count**.

📦 Here's what happens:

1. ✅ **OTA installs the update** into the **inactive slot** (for example, slot_b).
2. 🔄 **Bootloader switches** to that new slot and **tries to boot** it.
3. 📉 But sometimes the new update might fail to boot (maybe it crashes or freezes).

To avoid getting stuck:

✅ Retry Count: A Safety Net

- The retry count is like:
  "I will try to boot this slot 3 times before giving up."

So:

First Boot	Boot fails → Try again
Retry 3	Retry count becomes 2

Second Boot	Boot fails again → Try one last time
Retry 2	Retry count becomes 1

Third Boot	Boot fails again
Retry 1	Retry count becomes 0 → Give up

❌ What Happens if All Tries Fail?

- When `retry_count = 0`, bootloader says:

  > "This slot is broken, I'll mark it **invalid** and try the other slot!"

- Then it **boots from the other (older but working)** slot.

✅ What if Boot is Successful?

- If the boot is successful, Android tells the system:

  > "Mark this slot as **successful**."

- The retry count is no longer needed. Future boots go to this slot directly.

🔁 Summary Table

Boot Attempt	Retry Count	Outcome
First try	3	Boot fails → retry again

Boot Attempt	Retry Count	Outcome
Second try	2	Boot fails → retry again
Third try	1	Boot fails → retry ends
Retry = 0	0	Mark slot unbootable → switch slots
Boot success	Any	Mark slot as **successful**

Would you like a diagram to show this visually?

13. Lock and unlock the bootloader

bookmark_border
By default, most Android devices ship with a locked bootloader, meaning that users can't flash the bootloader or device partitions. If needed, you (and device users with Developer options enabled) can unlock the bootloader to flash new images.

Get into the bootloader
To issue fastboot commands, you must be in the bootloader mode. One way to do this is to send the adb command adb reboot bootloader. Some phones also allow you to reboot into the bootloader by pressing a key combination (commonly volume down) while booting the device.

Unlock the bootloader
Once in the bootloader mode, to unlock the bootloader and enable partitions to be reflashed, run the fastboot flashing unlock command on the device. After setting, the unlock mode persists across reboots.

Devices should deny the fastboot flashing unlock command unless the get_unlock_ability is set to 1. If set to 0, the user needs to boot to the home screen, open the Settings > System > Developer options menu and enable the OEM unlocking option (which sets the get_unlock_ability to 1). After setting, this mode persists across reboots and factory data resets.

When the fastboot flashing unlock command is sent, the device should prompt users to warn them that they might encounter problems with unofficial images. After the user acknowledges the warning, the device should perform a factory data reset to prevent unauthorized data access. The bootloader should reset the device even if it can't reformat it properly. Only after a reset can the persistent flag be set so that the device can be reflashed.

All RAM not already overwritten should be reset during the fastboot flashing unlock process. This measure prevents attacks that read leftover RAM contents from the previous boot. Similarly, unlocked devices should clear RAM at every boot (unless this creates an unacceptable delay), but should leave the region used for the kernel's ramoops.

Lock the bootloader
To lock the bootloader and reset the device, run the fastboot flashing lock command on the device. Devices intended for retail should be shipped in the locked state (with get_unlock_ability returning 0) to ensure that attackers can't compromise the device by installing a new system or boot image.

Set lock and unlock properties
The ro.oem_unlock_supported property should be set at build time based on whether the device supports flashing unlock.

If the device supports flashing unlock, set ro.oem_unlock_supported to 1.
If the device doesn't support flashing unlock, set ro.oem_unlock_supported to 0.
If the device supports flashing unlock, then the bootloader should indicate the lock status by setting the kernel command line variable androidboot.flash.locked to 1 if locked or to 0 if unlocked. This variable must be set in bootconfig instead of in the kernel command line in Android 12.

For devices that support dm-verity, use ro.boot.verifiedbootstate to set the value of ro.boot.flash.locked to 0; this unlocks the bootloader if the verified boot state is orange.

Protect critical sections
Devices should support the locking and unlocking of critical sections, which are defined as whatever is needed to boot the device into the bootloader. Such sections might include fuses, virtual partitions for a sensor hub, first-stage bootloader, and more. To lock critical sections, you must use a mechanism that prevents code (kernel, recovery image, OTA code, etc.) that is running on the device from deliberately modifying any critical section. OTAs should fail to update critical sections if the device is in lock critical state.

Transitioning from locked to unlocked state should require a physical interaction with the device. This interaction is similar to what the effects of running the fastboot flashing unlock command but requires the user to press a physical button on the device. Devices shouldn't allow programmatically transitioning from lock critical to unlock critical without physical interaction, and devices shouldn't ship in the unlock critical state.

Here's a **simple explanation** of how bootloader **locking and unlocking** works on Android devices:

🔐 What is the Bootloader?

The **bootloader** is the first thing that runs when your Android device turns on. It loads the system (like Android OS).

- A **locked bootloader** = secure and factory default.
- An **unlocked bootloader** = allows custom ROMs or system images to be flashed (but is less secure).

🧭 Steps to Unlock the Bootloader

1. **Enable Developer Options:**
   - Go to `Settings > About phone > Tap Build Number 7 times`.
   - Then go to `Settings > System > Developer options`.
2. **Enable OEM Unlocking:**
   - Turn on **OEM unlocking**.
   - This sets a flag (`get_unlock_ability = 1`) that allows unlocking.
3. **Reboot into Bootloader Mode:**
   - Run:

     ```nginx
     adb reboot bootloader
     ```
   - Or hold Volume Down + Power during boot.
4. **Unlock the Bootloader:**
   - Run this command:

     ```nginx
     fastboot flashing unlock
     ```
   - The device shows a warning: "Unlocking may cause issues."
   - If you confirm, it **factory resets** your device (wipes all data).
   - It clears all RAM to prevent data leaks.

🔒 How to Lock the Bootloader Again

1. **Boot into bootloader mode.**
2. **Run this command:**

   ```csharp
   fastboot flashing lock
   ```
3. It will **wipe the device again** for safety.

📜 Bootloader Status Properties

Property	Description
`ro.oem_unlock_supported`	Set to `1` if the device supports unlocking
`androidboot.flash.locked`	`1` if locked, `0` if unlocked (set via bootconfig)
`ro.boot.verifiedbootstate`	Set to `orange` if device is unlocked (used for `dm-verity`)

⚠️ Security Precautions

- **Factory Reset:** Happens during unlock/lock to erase all user data.
- **RAM wipe:** Clears RAM to prevent leftover data from being read.
- **Critical sections:** Like firmware or sensor hub can't be changed unless device is unlocked and user physically interacts (e.g., presses a button).

- **No programmatic unlocks:** Unlocking must require physical action—apps can't silently unlock.

🤔 Why Lock the Bootloader?

- **Security**: Prevents malicious OS or images from being installed.
- **DRM & Banking**: Many apps require locked bootloaders.
- **Retail devices** ship with the bootloader **locked by default**.

Would you like a flowchart or diagram to visualize the unlocking/locking process?

14. Move fastboot to userspace

bookmark_border
Fastboot is the name of a bootloader module and mode. Android 10 and higher supports resizable partitions by relocating the fastboot implementation from bootloader to userspace. This relocation enables moving the flashing code into a maintainable and testable common location with only the vendor-specific parts of fastboot implemented by a hardware abstraction layer (HAL). In addition, Android 12 and higher supports flashing ramdisks through an added fastboot command.

Unify fastboot and recovery
Because userspace fastboot and recovery are similar, you can merge them into one partition or binary. This provides advantages such as using less space, having fewer partitions overall, and having fastboot and recovery share their kernel and libraries.

Fastbootd is the name of a userspace daemon and mode. To support fastbootd, the bootloader must implement a new boot control block (BCB) command of boot-fastboot. To enter fastbootd mode, bootloader writes boot-fastboot into the command field of the BCB message and leaves the recovery field of BCB unchanged (to enable restarting any interrupted recovery tasks). The status, stage, and reserved fields remain unchanged as well. The bootloader loads and boots into the recovery image upon seeing boot-fastboot in the BCB command field. Recovery then parses the BCB message and switches to fastbootd mode.

ADB commands
This section describes the adb command for integrating fastbootd. The command has different results, depending on whether it's executed by system or by recovery.

Command	Description
reboot fastboot
Reboots into fastbootd (system).
Enters fastbootd directly without a reboot (recovery).
Fastboot commands
This section describes the fastboot commands for integrating fastbootd, including new commands for flashing and managing logical partitions. Some commands have different results, depending on whether they've been executed by bootloader or by fastbootd.

Command	Description
reboot recovery
Reboots into recovery (bootloader).
Enters recovery directly without a reboot (fastbootd).
reboot fastboot	Reboots into fastbootd.
getvar is-userspace
Returns yes (fastbootd).
Returns no (bootloader).
getvar is-logical:<partition>	Returns yes if the given partition is a logical partition, no otherwise. Logical partitions support all of the commands listed below.
getvar super-partition-name	Returns the name of the super partition. The name includes the current slot suffix if the super partition is an A/B partition (it usually isn't).
create-logical-partition <partition> <size>	Creates a logical partition with the given name and size. The name must not already exist as a logical partition.
delete-logical-partition <partition>	Deletes the given logical partition (effectively wipes the partition).
resize-logical-partition <partition> <size>	Resizes the logical partition to the new size without changing its contents. Fails if there isn't enough space available to perform the resize.
flash <partition> [<filename>] Writes a file to a flash partition. Device must be in the unlocked state.
erase <partition> Erases a partition (not required to be secure erase). Device must be in the unlocked state.
getvar <variable> | all	Displays a bootloader variable, or all variables. If the variable doesn't exist, returns an error.
set_active <slot>
Sets the given A/B booting slot as active. On the next boot attempt, the system boots from the specified slot.

For A/B support, slots are duplicated sets of partitions that can be booted from independently. Slots are named a, b, and so on, and differentiated by adding the suffixes _a, _b, and so on to the partition name.

reboot	Reboots device normally.
reboot-bootloader (or reboot bootloader)	Reboots device into bootloader.
fastboot fetch vendor_boot <out.img>
Use in Android 12 and higher to support flashing vendor ramdisks.

Gets the entire partition size and the chunk size. Gets data for each chunk, then stitches the data together to <out.img>

For details, see fastboot fetch vendor_boot <out.img>.

fastboot flash vendor_boot:default <vendor-ramdisk.img>
Use in Android 12 and higher to support flashing vendor ramdisks.

This is a special variant of the flash command. It performs a fetch vendor_boot image function, as if fastboot fetch was called. The new vendor_boot image it flashes depends on whether the boot header version is version 3 or version 4.

For details, see fastboot flash vendor_boot:default <vendor-ramdisk.img>.

fastboot flash vendor_boot:<foo> <vendor-ramdisk.img> Use in Android 12 and higher to support flashing vendor ramdisks.
Fetches the vendor_boot image. Returns an error if the vendor boot header is version 3. If it's version 4, it finds the correct vendor ramdisk fragment (if available). It replaces that with the given image, recalculates sizes and offsets, and flashes the new vendor_boot image.

For details, see fastboot flash vendor_boot:<foo> <vendor-ramdisk.img>

Fastboot and bootloader
The bootloader flashes the bootloader, radio, and boot/recovery partitions, after which the device boots into fastboot (userspace) and flashes all other partitions. The bootloader should support the following commands.

Command Description
download Downloads the image to flash.
flash recovery <image>/ flash boot <image>/ flash bootloader <image>/ Flashes recovery/boot partition and bootloader.
reboot Reboots the device.
reboot fastboot Reboots to fastboot.
reboot recovery Reboots to recovery.
getvar Gets a bootloader variable that is required for flashing of recovery/boot image (for example, current-slot and max-download-size).
oem <command> Command defined by OEM.
Dynamic partitions
The bootloader must not allow the flashing or erasing of dynamic partitions and must return an error if these operations are attempted. For retrofitted dynamic partition devices, the fastboot tool (and bootloader) supports a force mode to directly flash a dynamic partition while in bootloader mode. For example, if system is a dynamic partition on the retrofitted device, using the fastboot --force flash system command enables the bootloader (instead of fastbootd) to flash the partition.

Note: Force mode is intended to provide flexibility in factory flashing. AOSP doesn't recommend its use for developers.
Off-mode charging
If a device supports off-mode charging or otherwise autoboots into a special mode when power is applied, an implementation of the fastboot oem off-mode-charge 0 command must bypass these special modes, so that the device boots as if the user had pressed the power button.

Fastboot OEM HAL
To completely replace bootloader fastboot, fastboot must handle all existing fastboot commands. Many of these commands are from OEMs and are documented but require a custom implementation. Many OEM-specific commands aren't documented. To handle such commands, the fastboot HAL specifies the required OEM commands. OEMs can also implement their own commands.

The definition of fastboot HAL is as follows:

```
import IFastbootLogger;

/**
 * IFastboot interface implements vendor specific fastboot commands.
 */
interface IFastboot {
    /**
     * Returns a bool indicating whether the bootloader is enforcing verified
     * boot.
     *
     * @return verifiedBootState True if the bootloader is enforcing verified
     * boot and False otherwise.
     */
    isVerifiedBootEnabled() generates (bool verifiedBootState);

    /**
     * Returns a bool indicating the off-mode-charge setting. If off-mode
     * charging is enabled, the device autoboots into a special mode when
     * power is applied.
```

```
 *
 * @return offModeChargeState True if the setting is enabled and False if
 * not.
 */
isOffModeChargeEnabled() generates (bool offModeChargeState);

/**
 * Returns the minimum battery voltage required for flashing in mV.
 *
 * @return batteryVoltage Minimum battery voltage (in mV) required for
 * flashing to be successful.
 */
getBatteryVoltageFlashingThreshold() generates (int32_t batteryVoltage);

/**
 * Returns the file system type of the partition. This is only required for
 * physical partitions that need to be wiped and reformatted.
 *
 * @return type Can be ext4, f2fs or raw.
 * @return result SUCCESS if the operation is successful,
 * FAILURE_UNKNOWN if the partition is invalid or does not require
 * reformatting.
 */
getPartitionType(string partitionName) generates (FileSystemType type, Result result);

/**
 * Executes a fastboot OEM command.
 *
 * @param oemCmd The oem command that is passed to the fastboot HAL.
 * @response result Returns the status SUCCESS if the operation is
 * successful,
 * INVALID_ARGUMENT for bad arguments,
 * FAILURE_UNKNOWN for an invalid/unsupported command.
 */
doOemCommand(string oemCmd) generates (Result result);

};
```

Enable fastbootd
To enable fastbootd on a device:

Add fastbootd to PRODUCT_PACKAGES in device.mk: PRODUCT_PACKAGES += fastbootd.

Ensure that the fastboot HAL, boot control HAL, and health HAL are packaged as part of the recovery image.

Add any device-specific SEPolicy permissions required by fastbootd. For example, fastbootd requires write access to a device-specific partition to flash that partition. In addition, fastboot HAL implementation may also require device-specific permissions.

To validate userspace fastboot, run the Vendor Test Suite (VTS).

Flash vendor ramdisks
Android 12 and higher provides support for flashing ramdisks with an added fastboot command that pulls the full vendor_boot image from a device. The command prompts the host-side fastboot tool to read the vendor boot header, reimage, and flash the new image.

To pull the full vendor_boot image, the command fetch:vendor_boot was added to both the fastboot protocol, and the fastbootd implementation of the protocol in Android 12. Note that fastbootd does implement this, but the bootloader itself might not. OEMs can add the fetch:vendor_boot command to their bootloader implementation of the protocol. However, if the command isn't recognized in bootloader mode, then flashing individual vendor ramdisks in bootloader mode isn't a vendor-supported option.

Bootloader changes
The commands getvar:max-fetch-size and fetch:name are implemented in fastbootd. To support flashing vendor ramdisks in bootloader, you must implement these two commands.

Fastbootd changes
getvar:max-fetch-size is similar to max-download-size. It specifies the maximum size that the device can send in one DATA response. The driver must not fetch a size larger than this value.

fetch:name[:offset[:size]] performs a series of checks on the device. If all of the following are true, the fetch:name[:offset[:size]] command returns data:

The device is running a debuggable build.
The device is unlocked (boot state orange).
The fetched partition name is vendor_boot.
The size value falls within 0 < size <= max-fetch-size.
When these are verified, fetch:name[:offset[:size]] returns the partition size and offset. Note the following:

fetch:name is equivalent to fetch:name:0, which is equivalent to fetch:name:0:partition_size.
fetch:name:offset is equivalent to fetch:name:offset:(partition_size - offset)
Therefore fetch:name[:offset[:size]] = fetch:name:offset:(partition_size - offset).

When offset or partition_size (or both) are unspecified, the default values are used, which for offset is 0, and for size is the calculated value of partition_size - offset.

Offset specified, size unspecified: size = partition_size - offset
Neither specified: default values used for both, size = partition_size - 0.
For example, fetch:foo fetches the entire foo partition at offset 0.

Driver changes
Commands were added to the fastboot tool to implement driver changes. Each is linked to its full definition in the table of Fastboot commands.

fastboot fetch vendor_boot out.img

Calls getvar max-fetch-size to determine the chunk size.
Calls getvar partition-size:vendor_boot[_a] to determine the size of the entire partition.
Calls fastboot fetch vendor_boot[_a]:offset:size for each chunk. (The chunk size is greater than the vendor_boot size, so there's normally only one chunk.)
Stitches the data together, to out.img.
fastboot flash vendor_boot:default vendor-ramdisk.img

This is a special variant of the flash command. It fetches the vendor_boot image, as if fastboot fetch was called.

If the vendor boot is header version 3, it does the following:
Replaces the vendor ramdisk with the given image.
Flashes the new vendor_boot image.
If the vendor boot header is version 4, it does the following:
Replaces the whole vendor ramdisk with the given image so that the given image becomes the only vendor ramdisk fragment in the vendor_boot image.
Recalculates the size and offset in the vendor ramdisk table.
Flashes the new vendor_boot image.
fastboot flash vendor_boot:foo vendor-ramdisk.img

Fetches vendor_boot image, as if fastboot fetch was called.

If the vendor boot header is version 3, it returns an error.
If the vendor boot header is version 4, it does the following:

Finds the vendor ramdisk fragment with name ramdisk_<var><foo></var>. If not found, or if there are multiple matches, returns an error.
Replaces the vendor ramdisk fragment with the given image.
Recalculates each size and offset in the vendor ramdisk table.
Flashes the new vendor_boot image.
If <foo> isn't specified, it tries to find ramdisk_.

mkbootimg
The name default is reserved for naming vendor ramdisk fragments in Android 12 and higher. While the fastboot flash vendor_boot:default semantics remain the same, you must not name your ramdisk fragments as default.

SELinux changes
A change was made in fastbootd.te to support flashing vendor ramdisks.

Here's a **simple explanation** of why and how **Fastboot moved to userspace** in Android 10+ and what **fastbootd** is:

🧾 What is Fastboot?

Fastboot is a mode that lets you **flash partitions** (like boot, system) on your device — used mostly by developers, OEMs, and during factory programming or updates.

Traditionally, fastboot ran **inside the bootloader** (low-level firmware).

🚚 Why Move Fastboot to Userspace?

From Android 10 onwards:

Problem with bootloader fastboot	Userspace fastboot solves it by...
Hard to update or test	Moving flashing logic to Android userspace (normal Linux)
Device-specific code duplicated	Separating **common flashing logic** from **hardware-specific HALs**
Can't manage logical (dynamic) partitions well	Userspace fastboot (**`fastbootd`**) **understands dynamic partitions**

🧠 What is `fastbootd`?

- A **daemon running in userspace** (on Android recovery).
- It handles **flashing**, **dynamic partitions**, and **vendor ramdisks**.
- Uses the **same kernel and libraries** as recovery.

📦 Boot Modes Recap

Mode	Runs In	Can Do
Bootloader fastboot	Firmware	Flash basic partitions
fastbootd (userspace fastboot)	Android Recovery	Flash dynamic partitions, vendor ramdisks, more modern logic
Recovery	Android Recovery	Apply OTAs, factory reset, etc.

🔁 How does bootloader enter `fastbootd`?

1. Bootloader writes **`boot-fastboot`** in BCB (boot control block).
2. Recovery boots and sees this command.
3. Recovery enters **`fastbootd`** mode.

You can also do this manually:

```bash
adb reboot fastboot
```

🔧 Fastbootd Can Do More (New Commands)

Command	What it does
`create-logical-partition`	Create a dynamic partition
`resize-logical-partition`	Resize a partition
`delete-logical-partition`	Delete a partition
`flash vendor_boot:default`	Flash **vendor ramdisk** (Android 12+)
`fetch vendor_boot`	Pull complete vendor_boot image

🧱 Why It's Useful

1. 🔄 **OTA Updates**: Fastbootd supports modern OTA logic.

2. 🌡️ **Better Testing**: Flash logic is easier to validate (userspace, SEPolicy).
3. 📉 **Less Partition Bloat**: Recovery and fastbootd can share one image.
4. 💡 **More Flexible**: Resize/delete partitions dynamically (like on PC).

🏗️ Device Requirements

To use fastbootd:

- Add `fastbootd` to `PRODUCT_PACKAGES` in `device.mk`
- Include:
  - `fastboot HAL` (hardware-specific logic)
  - `boot_control HAL` (slot support)
  - `health HAL` (battery info)
- Grant proper SELinux permissions (`fastbootd.te`)
- Use `Vendor Test Suite (VTS)` to validate

⚠️ Important Notes

- Devices **must be unlocked** to use `flash` or `erase`.
- Fastbootd runs **only on debug/dev builds**.
- `fastboot --force` can flash dynamic partitions in bootloader (not recommended for devs).
- **Do not name vendor ramdisks as `default`** manually (reserved).

🔁 Flowchart Summary

1. You run `adb reboot fastboot`
2. Device enters recovery
3. Recovery sees "boot-fastboot" in BCB
4. It switches into **fastbootd**
5. Fastbootd listens for flash commands over USB (like `fastboot flash system`)

Would you like a diagram or a minimal working device.mk example to enable fastbootd?

15. ↪
🔁 How does bootloader enter `fastbootd`? -- 1. Bootloader writes `boot-fastboot` in BCB (boot control block). 2. Recovery boots and sees this command. 3. Recovery enters `fastbootd` mode.
explain in simple words

Let me explain in **very simple words** how the bootloader enters `fastbootd` mode:

🧠 What is happening?

You want the device to go into **userspace fastboot mode** (called `fastbootd`) — not the old bootloader fastboot.

🔁 Step-by-step:

1. **Bootloader writes a special message**:
   The bootloader saves a command called `boot-fastboot` into a tiny area of storage called **BCB** (Boot Control Block).
   Think of BCB like a notepad used to send messages between the bootloader and recovery.
2. **Bootloader starts the recovery system**:
   After writing the message, the bootloader **boots into the recovery image**, not into Android.
3. **Recovery checks that message**:
   Recovery reads the message in the BCB. If it sees `boot-fastboot`, it **doesn't open recovery UI**.
   Instead, it **starts a special program called `fastbootd`** — this is fastboot running inside Android userspace.

🧩 Why is this useful?

Because `fastbootd`:

- Runs with more features than bootloader fastboot.
- Can flash **dynamic partitions**.
- Is **easier to update and maintain**.

🧪 Summary:

It's like:

> Bootloader leaves a sticky note 📝 for recovery: "Hey, don't open recovery. Go into fastbootd mode!"
>
> Recovery reads it, nods 👌, and enters fastbootd instead of showing the normal recovery screen.

Would you like a **visual diagram** for this flow too?

16. Generic Bootloader (GBL) overview

bookmark_border
The Generic Bootloader (GBL) is a standardized, updatable bootloader solution designed to streamline the Android boot process. GBL replaces the highly fragmented landscape of vendor-specific bootloaders with a single, consistently maintained component. This change reduces redundancy, improves security, and simplifies updates across the Android ecosystem.

GBL consists of the following components:

core Android boot logic
The logic that includes the main program loop, boot mode detection, and loading the kernel.
Fastboot
A communication protocol and diagnostic tool used for device flashing and debug.
Vendor extensions
Extension of the aforementioned components of GBL by creating your own protocols. These protocols are agreed upon in advance before GBL adds them to their dependency list. Vendor extensions are optional.
UEFI protocol handlers
Implementations for required and recommended UEFI protocols, including block I/O, memory allocation, and random number generation. Reference implementations are found in the boot firmware projects (EDK2, UBoot, LK).
Android Specific UEFI Protocol definitions
Custom UEFI protocols for AVB, Fastboot, slot selection, OS configuration, and more. Implementations are found in Android Silicon Vendor firmware projects.
Beginning with Android 16, if you ship a device based on ARM-64 chipset, we strongly recommend that you deploy the latest Google-signed version of GBL and integrate it into your boot chain.

What's next?
To learn how to deploy the Generic Bootloader, refer to Deploy gbl.

Was this helpful?

Here's a **simple explanation** of the **Generic Bootloader (GBL)** in Android:

🔧 What is the Generic Bootloader (GBL)?

The **Generic Bootloader (GBL)** is a **standardized bootloader** designed by Google to **replace vendor-specific bootloaders** on Android devices.

🧠 Why do we need GBL?

Before GBL:

- Every company (Samsung, Xiaomi, etc.) had their own **custom bootloaders**.
- These were often:
    - Hard to maintain 😩
    - Not always secure 🔓
    - Incompatible with new Android features ❌

With GBL:

- Everyone can use a **common, secure, and updateable bootloader** ✅
- Saves time, reduces bugs, and improves security 🔐

📦 What does GBL include?

1. **Core Android boot logic**
    - Figures out the boot mode (normal, recovery, fastboot, etc.)
    - Loads the kernel

2. **Fastboot**
   - The standard tool to flash devices and run debug commands
3. **Vendor extensions** (optional)
   - Manufacturers can add their own special commands
   - These are agreed in advance and plugged into GBL
4. **UEFI protocol handlers**
   - UEFI = the modern boot standard used in PCs and now Android
   - Handles things like memory, storage, random numbers
5. **Android-specific UEFI protocols**
   - For things like Verified Boot (AVB), slot selection (A/B), OS boot config

---

📢 Starting from Android 16:

If your device uses an **ARM64 processor**, Google **recommends**:

> Use the latest **Google-signed version of GBL** as your bootloader.

This ensures:

- Better compatibility
- Easier updates
- Higher trust/security

✅ Benefits of using GBL:

Feature	Benefit
✅ Standardized	Less fragmentation
🔐 Secure	Google-reviewed bootloader
🔁 Updateable	Easier OTA maintenance
🧩 Modular	Vendors can add safe extensions
🚀 Supports UEFI	Future-ready and clean interface

🧭 What's next?

If you're building or modifying Android devices, your next step is:
👉 **Learn how to deploy GBL** using Google's guide to `Deploy gbl` (the link they refer to in the doc).

Would you like a **visual architecture diagram** of GBL and its components too?

17. ↪
UEFI protocol
explain in simple words

Sure! Let's explain **UEFI protocol** in **simple words**, especially how it relates to Android and GBL:

🔧 What is UEFI?

UEFI (Unified Extensible Firmware Interface) is like the **brain of the boot process** in modern computers and phones.

Think of it as a **standard set of rules** that tells your device **how to start up** (boot) and **load the operating system**.

Before UEFI, we had **BIOS**, which was older and limited.

🧠 Why use UEFI in Android?

Google and Android are adopting UEFI in things like **GBL (Generic Bootloader)** to:

- Make booting **faster and more secure** 🔐
- Allow Android to run on **more devices** (like laptops, tablets, embedded systems)
- Support **modern boot features** like Verified Boot, Fastboot, and Slot A/B logic

---

📦 What is a UEFI Protocol?

A **UEFI protocol** is just a **set of instructions** (like a library or interface) for doing specific jobs during boot.

Examples of jobs (protocols):

Protocol Type	What It Does
Block I/O Protocol	Reads/writes to storage (e.g. eMMC, UFS)
Memory Allocation	Allocates memory before Android starts
RNG (Random Number Gen)	Generates secure random numbers
Android Boot Config	Chooses which slot (A or B) to boot
AVB Protocol	Verifies the boot partition (security)
Fastboot Protocol	Enables flashing over USB

🤝 How Android Uses It

When Android boots with **GBL + UEFI**, it uses these protocols like building blocks:

- GBL says: “I need to load the kernel”
- UEFI’s block I/O protocol says: “Okay, here’s how to read it from disk”
- GBL says: “Is this kernel trusted?”
- UEFI AVB protocol replies: “Yes, verified!”

It’s all modular and reusable, like **LEGO blocks** that all speak the same language 🧱

✅ In short:

UEFI protocols = standard tools used during boot to do key things like memory, storage, security checks, etc.

They're now being used in Android (starting with Android 16 and GBL) to make booting:

- **More modern**
- **More secure**
- **More consistent across devices**

Would you like a **simple diagram or table** that shows how GBL + UEFI work together?

18. Deploy GBL

bookmark_border
This page explains how to deploy the GBL binary.

Boot firmware requirements
To use GBL, the boot firmware must meet the following requirements:

UEFI compliance. The firmware must implement and use the required UEFI protocols. The firmware must also allow for vendor-specific extensions using defined UEFI protocols.

Security. The firmware must implement all aspects of Android Verified Boot (AVB), ensuring only authenticated images are loaded.

Boot modes. The binary should be able to handle various boot modes, such as normal boot, recovery boot, and fastboot.

Dynamic partitioning. The boot firmware must implement slot selection logic so that it supports reading the correct A/B boot slot and is compatible with dynamic partitions and userdata in super.

OS configuration. The firmware must be capable of modifying the kernel command line, device tree (DTB), and bootconfig with OEM customizations needed to boot the device.

Protected VM loading. The binary should correctly load preverified protected VM firmware before the Android kernel in the presence of protected VMs. For further information, see Microdroid boot sequence.

Memory management. The boot firmware must support the UEFI memory allocation API.

Compatibility and backward compatibility. The firmware should work on devices with different vendor, SOCs, and maintain backward compatibility with the corresponding Android version.

Boot firmware support
With the modifications necessary to support requirements in the previous section, the following UEFI firmware implementations work with the GBF:

EDK2 (Tianocore). A EDK2 is a popular open-source UEFI implementation. GBL support is needed for EDK2-based bootloaders, and UEFI support is already present.
U-Boot. A flexible and widely used open-source bootloader project that is gaining UEFI compatibility for GBL usage.
LittleKernel (LK). An open-source bootloader used by some vendors.
Run GBL
You can obtain a prebuilt GBL binary to run or build your own and run it.

Obtain and run the GBL binary
GBL is distributed as a single EFI app binary. You can update this binary independently from the device's base firmware using Android's standard update mechanism.

Beginning with Android 16, if you ship a device based on ARM-64 chipset, we strongly recommend that you deploy the latest Google-signed version of GBL and integrate it into your boot chain.

Build and run the GBL
To build and run the GBL:

Verify that you have the repo tool and Bazel bootstrap installed:

```
sudo apt install repo bazel-bootstrap
```
Initialize your current directory for source control using the uefi-gbl-mainline manifest file:

```
repo init -u https://android.googlesource.com/kernel/manifest -b uefi-gbl-mainline
repo sync -j16
```
Build the EFI app:

```
./tools/bazel run //bootable/libbootloader:gbl_efi_dist --extra_toolchains=@gbl//toolchain:all
```

Run the EFI app within Cuttlefish:

cvd start --android_efi_loader=path_to_the_EFI_app ...
Instead of booting Android directly, this cvd start command uses the EFI app to boot Android.

Note: For x86 platform, use the EFI image built for x86_64.
File bugs and contact the bootloader team
To report a bug for the GBL, navigate to the Android Generic Bootloader component in Buganizer.

For questions, contact the GBL team, send an email to android-gbl@google.com.

Was this helpful?

Here's a **simple explanation** of how to **deploy the Generic Bootloader (GBL)** in Android:

🚀 What Is GBL?

Generic Bootloader (GBL) is a standardized Android bootloader from Google that replaces custom vendor bootloaders.
It uses **UEFI** (like modern PCs) to simplify boot, updates, and improve security 🔐.

📦 What You Need to Use GBL

To deploy GBL, your device's **boot firmware** must support:

✅ Requirement	📝 What It Means
UEFI Support	Must support standard UEFI protocols (like memory, storage, random number, etc.)
Security	Must enforce **AVB (Android Verified Boot)** to ensure only signed images can run
Boot Modes	Must support **normal**, **recovery**, and **fastboot** modes
Dynamic Partitioning	Must support **A/B slot logic** and dynamic partitions (like **super**)
OS Configuration	Must be able to **modify kernel cmdline**, DTB, bootconfig at boot
Protected VM Boot	If using protected virtual machines (e.g. Microdroid), GBL should load their firmware first
UEFI Memory APIs	Must support UEFI memory management functions
Backwards Compatibility	Should work across various SoCs and Android versions

---

🧰 Supported Bootloaders

You can use GBL with popular open-source boot firmware:

Bootloader	Notes
EDK2 (Tianocore)	Official UEFI implementation, works well with GBL
U-Boot	Widely used, now gaining UEFI support
LK (LittleKernel)	Used by some vendors, may need customization

---

🛠️ How to Use GBL

You have **2 options**:

✅ 1. Use Google's Prebuilt GBL

- Google provides a **signed EFI binary** (`gbl.efi`)
- You **flash it like any other image** during OTA update
- Recommended for Android 16+ and ARM64 devices

⚙️ 2. Build GBL Yourself

Step-by-step:

1. **Install required tools**

```bash
sudo apt install repo bazel-bootstrap
```

2. **Initialize repo**

```bash
repo init -u https://android.googlesource.com/kernel/manifest -b uefi-gbl-mainline
repo sync -j16
```

3. **Build the GBL EFI binary**

```bash
./tools/bazel run //bootable/libbootloader:gbl_efi_dist --extra_toolchains=@gbl//toolchain:all
```

4. **Run GBL inside emulator (Cuttlefish)**

```bash
cvd start --android_efi_loader=path_to_the_EFI_app
```

> ⚠️ If you're using **x86**, make sure to build an **x86_64** GBL binary.

🐛 Reporting Issues

- File bugs: Android GBL in Buganizer
- Email questions: android-gbl@google.com

✅ Summary

Step	What to Do
1️⃣	Make sure your bootloader supports UEFI, AVB, A/B
2️⃣	Choose prebuilt or build your own GBL
3️⃣	Flash it as **EFI** app using OTA or during factory setup
4️⃣	Test it in Cuttlefish before device deployment

Let me know if you'd like a **diagram** showing how GBL fits into the Android boot process.

19. how GBL worked UEFI
explain in simple words

Here's a **simple explanation** of how **GBL (Generic Bootloader)** works with **UEFI**:

🧠 First, What is UEFI?

UEFI (Unified Extensible Firmware Interface) is like the "brain" that runs **before** Android starts.
It's the **modern replacement for BIOS**, used to:

- Load bootloaders or operating systems
- Manage memory, storage, and devices
- Provide secure boot features

🤝 What is GBL?

GBL = Generic Bootloader

It's a **universal, Google-made bootloader** that uses **UEFI protocols** instead of every phone having its own custom bootloader.

Think of GBL like a "universal boot menu" that:

- Talks to Android using UEFI language
- Boots Android safely
- Updates itself easily

🔗 How GBL Works with UEFI (Step by Step)

Step	What Happens
1. Device Turns On	The system firmware (like EDK2 or U-Boot) runs — it's UEFI-compliant
2. UEFI Loads GBL	UEFI loads the **GBL EFI binary (`gbl.efi`)** like a program
3. GBL Starts	GBL runs and uses UEFI APIs to talk to storage, memory, and devices
4. GBL Selects Boot Mode	GBL checks: normal boot, recovery, or fastboot
5. GBL Chooses Slot (A/B)	GBL reads metadata and selects which slot (A or B) to boot from
6. GBL Prepares Kernel	GBL modifies kernel command line, bootconfig, or device tree if needed
7. GBL Starts Android Kernel	GBL hands control to the Android kernel to start Android OS

✅ Why This UEFI + GBL Setup Is Great

Benefit	Why It Matters
🔒 Security	UEFI + AVB (Verified Boot) ensures only trusted images run
♻️ Easier Updates	GBL can be updated via OTA like any normal image
🛠️ Simpler for Vendors	No need to write their own bootloader logic
🤖 Android Standard	GBL is maintained by Google — consistent and reliable

🎯 Real-World Analogy

Imagine UEFI is the **platform/OS** in a smart TV.
GBL is the **Netflix app** that always works no matter the TV brand, because it follows the **same rules (UEFI protocols)**.

Let me know if you'd like a **diagram** of how GBL fits into the boot flow.

20. Now tell me full boot flow in android as per various changes in android over the time

Here's a full **Android boot flow**, including how it evolved across Android versions. I'll break it down into **core stages**, then summarize the **evolution** and **key changes** over time (e.g. A/B updates, GKI, fastbootd, GBL, etc.).

🔁 Overview of Android Boot Flow

✅ 1. Power-On & ROM Execution

- **Who:** SoC vendor-provided **Boot ROM (BROM)**
- **What:** First code that executes from a non-writable area (hardcoded into silicon)
- **Job:**
  - Initializes secure environment (TrustZone / Secure Monitor)
  - Loads **1st-stage bootloader** (e.g., **SPL**, PBL) from storage

---

✅ 2. 1st Stage Bootloader (SPL/PBL)

- Very minimal
- Verifies and loads the **2nd stage bootloader**
- Typically stored in bootloader partition or fuse-protected storage

---

✅ 3. 2nd Stage Bootloader

- Examples: U-Boot, Little Kernel (LK), Qualcomm Aboot, EDK2 (UEFI), etc.
- **Job:**
  - Load the **DTB**, **kernel**, **vendor boot**, etc.
  - Initialize **fastboot**, USB, display, memory
  - Support **Verified Boot (AVB)** for integrity check
  - Handle **A/B slot selection** if enabled
  - Load the correct **boot/recovery** partition

✅ 4. Fastboot / Fastbootd (if triggered)

- Fastboot (bootloader mode): CLI interface for flashing/debugging
- **Android 10+**: Fastboot moved to **userspace (fastbootd)**
  - For flashing dynamic partitions (like super, system, vendor)
  - Merged with **recovery image**

✅ 5. Kernel Boot

- Kernel + DTB + Bootconfig are loaded
- **Android Verified Boot (AVB)** checks images' authenticity
- Kernel initializes memory, drivers, mounts rootfs

✅ 6. Initramfs / Ramdisk

- Extracted from **boot** or **vendor_boot** partition
- Runs **init** binary (first userspace process)
- Parses `init.rc`, sets up system daemons, SELinux policies

✅ 7. Init & Zygote Start

- `init` launches **Zygote**, which starts the **Android Runtime (ART)**
- **Zygote** spawns:
  - **SystemServer**: starts system services
  - **SurfaceFlinger**: manages display
  - **App_process**: manages apps

✅ 8. Boot Completed

- System ready
- **Launcher** starts
- User can interact with the system

🧭 Evolution of Android Boot Flow by Version

Android Version	Key Boot Change
<= Android 7	No A/B updates, separate recovery partition, static partitions
Android 8 (Oreo)	Introduced **A/B (Seamless Updates)** & **Boot Control HAL**
Android 9	Support for **Dynamic Partitions** and **VNDK** separation
Android 10	**Fastbootd** introduced (userspace fastboot); DTB moved into boot partition
Android 11	**GKI** (Generic Kernel Image); Vendor Boot partition added
Android 12	**Bootconfig** replaces kernel cmdline; Improved ramdisk handling
Android 13+	Continued modularization: system_ext, product partitions
Android 14+	Enhanced **Protected VM** support; **Preverified microdroid boot**
Android 16 (upcoming)	Strongly recommends **Generic Bootloader (GBL)** using **UEFI**

🔐 Verified Boot (AVB)

- Ensures all boot images (boot, vendor_boot, vbmeta) are cryptographically verified
- Uses `vbmeta` partition
- Enforces **boot state**: green (verified), orange (unlocked), red (tampered)

📦 Partitions Involved (Modern Devices)

- **boot**: Kernel + initramfs
- **vendor_boot**: vendor-specific ramdisk (since Android 11)
- **vbmeta**: AVB metadata
- **misc**: OTA communication (e.g., boot to recovery)
- **dtbo**: Device Tree Overlays (if needed)
- **super**: Holds logical partitions (system, vendor, product, etc.)
- **recovery**: Still used in non-A/B devices or fastbootd

📊 Summary of What Changed Over Time

Feature	Introduced In	Description
A/B Slots	Android 8	Enables seamless updates
Dynamic Partitions	Android 9+	System, vendor etc. merged into “super” partition
Fastbootd	Android 10	Moved fastboot functionality into recovery (userspace)

Feature	Introduced In	Description
GKI & vendor_boot	Android 11	Kernel is common; vendor-specific things moved to vendor_boot
Bootconfig	Android 12	Replaces kernel cmdline for cleaner config
GBL	Android 16	Standardized UEFI-based bootloader replacing fragmented vendor bootloaders

Would you like a **diagram** of the full modern Android boot flow with A/B, AVB, GBL, fastbootd etc.?