Kernel_AOSP-1

1. where driver code is located in android. 5
2. In which locations kernel module is placed in android. 7
3. confusing. 9
4. types of modules here. 11
5. Built-in vs Loadableexplain this in simple words. 13
6. Inside the kernel?which partition and which directory in source code. 14
7. which which partion's modules are loadable and builtin from which directory. 16
8. ok explain dt. 18
9. how many dts are there in android f. 20
10. device variantthis. 22
11. a phone with different ram sizes, could be a varient. 24
12. yes. 26
13. dts vs dtsi vs dtb. 28
14. ok explain dts and dtsi with sample. 30
15. please explain the content as well. 32
16. but here i can't see link between kernel modules with this. 34
17. confused. 36
18. can modules talk to each other?. 38
19. explain in details. 40
20. Yes. 43
21. Syntax of dts. 46
22. `reg = <addr size>`what is this. 48
23. 50
24. The Android kernel is based on an upstream Linux Long Term Supported (LTS) kernel. At Google, LTS kernels are combined with Android-specific patches to form what are known as Android Common Kernels (ACKs). Newer ACKs (version 5.4 and above) are also known as GKI kernels. GKI kernels support the separation of the hardware-agnostic generic core kernel code and GKI modules from the hardware-specific vendor modules. The GKI kernel interacts with hardware-specific vendor modules containing system on a chip (SoC) and board-specific code. The interaction between the GKI kernel and vendor modules is enabled by the Kernel Module Interface (KMI) consisting of symbol lists identifying the functions and global data required by vendor modules. Figure. 52
25. **KMI (Kernel Module Interface)**explain simple words. 54
26. KMI (Kernel Module Interface)why we use this and the benefits. 55
27. GKI modules vs Vendor modules. 57
28. Kernel glossary This section contains terms used throughout the kernel documentation. Kernel types Android Common Kernel (ACK) A kernel that is downstream of a LTS kernel and includes patches of interest to the Android community that haven't been merged into Linux mainline or Long Term Supported (LTS) kernels. Newer ACKs (version 5.4 and above) are also known as GKI kernels as they support the separation of hardware agnostic Generic Kernel code and hardware agnostic GKI modules. Android Open Source Project (AOSP) kernel See Android Common Kernel. Feature Kernel A kernel for which the platform release features are guaranteed to be implemented. For example, in Android 12, the two feature kernels were android12-5.4 and android12-5.10. Android. 59
29. can you types of kernel u mentioned and why they exist?. 62
30. still confused. 66
31. **Feature Kernel**explain with example. 68
32. Boot time optimization bookmark_border This page provides tips to improve boot time. Strip debug symbols from modules Similar to how debug symbols are stripped from the kernel on a production device, make sure you also strip the debug symbols from modules. Stripping debug symbols from modules helps boot time by reducing the following: The time it takes to read the binaries from flash. The time it takes to decompress the ramdisk. The time it takes to load the modules. Stripping debug symbol from modules may save several seconds during boot. Symbol stripping is enabled by default in the Android platform build, but to explicitly enable them, set BOARD_DO_NOT_STRIP_VENDOR_RAMDISK_MODULES in your device-specific config under device/vendor/de. 70
33. explain in simple words. 77
34. Debug features bookmark_border The Linux kernel provides many mechanisms to diagnose issues including tracing, BPF, and profiling. Most of these techniques are available in the Generic Kernel Image (GKI) binary kernel. When developing products, there are many more tools that are integrated into the development or integration processes, some of which require kernel features that aren't part of GKI. As with most GKI features, we encourage you to work with upstream maintainers to add these features to Linux. Note: Kernel debugging tools and techniques are beyond the scope of this document. Downstream debug features in vendor modules Debugging features are similar in scope to other kinds of features and must follow the same processes as descr. 79
35. yes. 81
36. Kernel release notes bookmark_border This page summarizes the major features in each kernel release and provides links to additional information. Kernel 6.12 new features This section explains new features in kernel 6.12. Memory allocation profiling The key to understanding memory use is knowing where allocations are made. Kernel 6.12 includes a new memory allocation attribution system called memory allocation profiling (CONFIG_MEM_ALLOC_PROFILING in the config). With memory allocation profiling, each allocation is attributed to a unique source line so that issues with allocations can be identified quickly. Additionally, memory allocation profiling: Is used during the engineering phase, but is available in the standard GKI image. Can b. 83
37. Stable kernel releases and updates bookmark_border The Linux kernel stable release model started in 2005, when it was determined that the existing kernel development model (a new release every 2-3 months) was not meeting the needs of most users. Users wanted bugfixes made during those 2-3 months, and Linux distributions found it difficult to keep kernels up to date without feedback from the kernel community. In general, attempts to keep individual kernels secure and with the latest bugfixes was a large and confusing effort by lots of different individuals. Stable kernel releases are based directly on Linus Torvalds' releases, and are released every week or so, depending on various external factors (time of year, available patches, maintai. 86
38. difference between ACK and GKI in simple words. 90
39. GKI development bookmark_border This guide provides a high-level overview of GKI development. Review existing documentation Before you begin GKI development, read the following documents: For an overview of the GKI project, read GKI project. For an explanation of how Android Common Kernels (ACKs), GKI kernels, and KMI relate, read Android Common Kernels (ACK). To learn how to contribute to the kernel, read Developing kernel code for GKI, Contribution guidelines for Android Common Kernels, and Linux kernel coding standards. To learn about GKI and vendor modules, read Kernel modules overview and other documents in the modules section. Build, monitor, and test When building, monitoring, and testing your GKI implementations, follow these. 92
40. explain in simple words. 95
41. LLVM & hermeticexplain in details in simpl words. 97
42. Hermeticconfusing. 99
43. so clang will be included here in hermetic?. 101

44. GKI versioning scheme bookmark_border This page describes the versioning scheme for Generic Kernel Images (GKIs). A Generic Kernel Image (GKI) has a unique identifier called the kernel release. The kernel release consists of the kernel module interface (KMI) version and the sublevel. The kernel release is specific to the image being released, whereas the KMI version represents the interface that a release is built from. A KMI version can support multiple kernel releases. A kernel release is tied to only one KMI version. In the unlikely event where the kernel module interface has to be changed, the KMI generation is iterated to reflect the change in KMI version. Summary of terms The following table summarizes important terms used on this p. 102

45. Generic Kernel Image (GKI) release builds bookmark_border This page provides a listing of different versions of GKI release builds. Execute the following to sync the kernel source code: repo init -u https://android.googlesource.com/kernel/manifest mv <kernel_manifest.xml> .repo/manifests repo init -m manifest.xml repo sync Releases Refer to the following for information on specific release builds: android16-6.12 Releases android15-6.6 Releases android14-6.1 Releases android14-5.15 Releases android13-5.15 Releases android13-5.10 Releases android12-5.10 Releases. 106

46. ### 🔧 How to Sync GKI Source Code To fetch GKI source code for a specific release: bash CopyEdit `repo init -u https://android.googlesource.com/kernel/manifest mv <kernel_manifest.xml> .repo/manifests # rename your manifest repo init -m manifest.xml # use the renamed manifest repo sync # fetch all kernel source`make it simple. 108

47. why manifest here?. 109

48. Generic Kernel Image (GKI) release process bookmark_border This page describes how GKI is released, including weekly, quarterly, and out of band emergency releases. The goal of this document is to give OEMs a guideline on where to pick up the GKI as well as the process for out of band emergency fixes. OEMs can also use GKI development to learn more about how they can work with the Android Kernel team to optimize the GKI kernel for their products. GKI release cadence GKI is released on a quarterly cadence post KMI freeze. Release Month a12-5.10 a13-5.10 a13-5.15 a14-5.15 a14-6.1 a15-6.6 a16-6.12 June 2025 Check-in cut off GKI preload ready Jun 16 Jun 30 Jun 2 Jun 16 Jun 2 Jun 16 Jun 2 Jun 18 July 2025 Check-in cut off GKI preload ready. 110

49. Where do OEMs get GKI builds?explain more. 113

50. Quarterly certified releases GKI quarterly releases contain a tested boot.img that includes a Google inserted certificate to attest that the binaries were built from a known source code baseline. Each quarter, a GKI quarterly release candidate (not certified) is selected after the check-in cut off date, which is usually the second weekly build of that month. After the quarterly release candidate is selected, new changes won't be accepted into that month's release. During the closed window period, only fixes for bugs that cause test failure can be addressed. The release candidate undergoes quality assurance—as described in the GKI qualification section—to ensure compliance tests pass on GSI+GKI build with a reference device as well as cuttl. 115

51. Submit a respin request The following diagram shows the respin process. The process begins when the OEM Partner (you) submits the respin request. To enter into the respin process: Fill out the GKI Respin request form. and reach out to your Google Technical Account Manager immediately. This form creates a GKI respin request bug. Respin request bugs are visible to you (the requester), the GKI team, and specific individuals that you add to the bug's CC list. If you already have a fix, the request should point to the patch submittal in AOSP so Google can review it. If submitting the patch isn't feasible, the patch must be attached as a text file to the request. If you don't have a fix, the request must contain as much information as possible,. 118

52. Maintain a stable kernel module interface bookmark_border It's critical to maintain a stable kernel module interface (KMI) for vendor modules. The GKI kernel is built and shipped in binary form and vendor-loadable modules are built in a separate tree. The resulting GKI kernel and vendor modules must work as though they were built together. Generally, the Linux community has frowned on the notion of in-kernel ABI stability for the mainline kernel. In the face of different toolchains, configurations, and an ever-evolving Linux mainline kernel, it isn't feasible to maintain a stable KMI in mainline. However, it's possible to maintain a stable KMI in the highly-constrained GKI environment with these constraints: Only a single configuration,. 122

53. why ABI matter here? explain in simple. 125

54. ABI vs KMI. 127

55. Android kernel ABI monitoring bookmark_border You can use application binary interface (ABI) monitoring tooling, available in Android 11 and higher, to stabilize the in-kernel ABI of Android kernels. The tooling collects and compares ABI representations from existing kernel binaries (vmlinux+ GKI modules). These ABI representations are the .stg files and the symbol lists. The interface on which the representation gives a view is called the Kernel Module Interface (KMI). You can use the tooling to track and mitigate changes to the KMI. The ABI monitoring tooling is developed in AOSP and uses STG (or libabigail in Android 13 and lower) to generate and compare representations. This page describes the tooling, the process of collecting and a. 129

56. Resolve kernel ABI breakages You can handle kernel ABI breakages by refactoring the code to not change the ABI or updating the ABI representation. Use the following chart to determine the best approach for your situation. Refactor code to avoid ABI changes Make every effort to avoid modifying the existing ABI. In many cases, you can refactor your code to remove changes that affect the ABI. Refactoring struct field changes. If a change modifies the ABI for a debug feature, add an #ifdef around the fields (in the structs and source references) and make sure the CONFIG used for the #ifdef is disabled for the production defconfig and gki_defconfig. For an example of how a debug config can be added to a struct without breaking the ABI, refer. 133

57. Refactor code to avoid ABI changes Make every effort to avoid modifying the existing ABI. In many cases, you can refactor your code to remove changes that affect the ABI. Refactoring struct field changes. If a change modifies the ABI for a debug feature, add an #ifdef around the fields (in the structs and source references) and make sure the CONFIG used for the #ifdef is disabled for the production defconfig and gki_defconfig. For an example of how a debug config can be added to a struct without breaking the ABI, refer to this patchset. Refactoring features to not change the core kernel. If new features need to be added to ACK to support the partner modules, try to refactor the ABI part of the change to avoid modifying the kernel ABI. For. 141

58. Run ABI monitoring bookmark_border This page describes how to build Android kernel ABI representations and run ABI monitoring. It's applicable to Android 16 and higher. For lower versions, refer to ABI monitor previous kernel versions. Also see the reference documentation for Kleaf: Support ABI monitoring (GKI) and Support ABI monitoring (Device). Build the kernel and its ABI representation After downloading GKI sources run the following command to build the GKI kernel and ABI artifacts: tools/bazel run //common:kernel_aarch64_abi_dist This command builds current ABI representation and copies it to $DIST_DIR/abi.stg along with the built kernel and modules. $DIST_DIR defaults to out_abi/kernel_aarch64_abi_dist/dist. You can specify ext. 151

59. what exactly is ABI explain with example. 155

60. Work with symbol lists bookmark_border To reduce the surface of symbols and types that need to be maintained as stable, the GKI kernel has capabilities to limit exported symbols to only those that are needed by modules. For externally compiled modules, you need to have a list of used symbols to allow them to be exported by the GKI kernel. For example, symbols used by modules for Cuttlefish are stored in gki/aarch64/symbols/virtual_device. Add a target for the symbol list generation Symbol lists are generated by the kernel_abi target. Add this target to the device BUILD.bazel with the following options: name Should be in the format of <kernel_build>_abi. kernel_build Should contain the name of the device kernel_build target. You can a. 158

61. Kernel modules overview bookmark_border There are two types of kernel modules: hardware agnostic GKI modules and hardware-specific vendor modules. This page provides an overview of both types of modules. GKI modules Generic kernel image (GKI) modules are used to deliver nonboot-required kernel capabilities separate from the generic core kernel. With GKI modules, you can choose specific kernel capabilities to use, often reducing kernel image size and runtime memory consumption. The reduction in size makes GKI well suited for Android Go devices and other resource-restricted form factors. GKI modules also provide a mechanism to let vendors incorporate new upstream features after the KMI freeze milestone. Built-in code can't be replaced with. 163

62. Configure kernel features as GKI modules bookmark_border This page covers how to configure a new kernel feature as a GKI module or configure an existing built-in kernel feature as a GKI module. Note: Changes that result in a new GKI module or that change the protection status of a GKI module must be approved by Google. Configure a new feature as a GKI module For the new feature, edit gki_defconfig and set the required kernel feature's config item from n to m (=m). Set this setting in both arch/arm64/configs/gki_defconfig and arch/x86/configs/gki_defconfig. Add the KO (.ko) files generated for the feature to the COMMON_GKI_MODULES_LIST section of common/modules.bzl. Add the files in a sorted order. If you're unsure of all the files genera. 166

63. Vendor module guidelines bookmark_border Use the following guidelines to increase the robustness and reliability of your vendor modules. Many guidelines, when followed, can help make it easier to determine the correct module load order and the order in which drivers must probe for devices. Note: Guidelines are listed in the order of importance and reflect the frequency of errors found in vendor modules. A module can be a library or a driver. Library modules are libraries that provide APIs for other modules to use. Such modules typically aren't hardware-specific. Examples of library modules include an AES encryption module, the remoteproc framework that's compiled as a module, and a logbuffer module. The module code in module_init() runs. 171
64. Loadable kernel modules bookmark_border As part of the module kernel requirements introduced in Android 8.0, all system-on-chip (SoC) kernels must support loadable kernel modules. Kernel configuration options To support loadable kernel modules, android-base.config in all common kernels includes the following kernel-config options (or their kernel-version equivalent): CONFIG_MODULES=y CONFIG_MODULE_UNLOAD=y CONFIG_MODVERSIONS=y All device kernels must enable these options. Kernel modules should also support unloading and reloading whenever possible. Note: CONFIG_MODULE_SRCVERSION_ALL is optional and isn't tested against. Module signing Module-signing is not supported for GKI vendor modules. On devices required to support verified boot,. 179
65. `modprobe`what is this. 183
66. Kernel module support bookmark_border A generic kernel image (GKI) may not contain the required driver support to enable a device to mount partitions. To enable a device to mount partitions and to continue booting, first-stage init is enhanced to load the kernel modules present on a ramdisk. The ramdisk is split into generic and vendor ramdisks. Vendor kernel modules are stored in the vendor ramdisk. The order in which kernel modules are loaded is configurable. Module location The ramdisk is the filesystem for first-stage init, and for the recovery/fastbootd image on A/B and virtual A/B devices. It's an initramfs composed of two cpio archives that get concatenated by the bootloader. The first cpio archive, which is stored as the vendor ra. 185
67. what is full android here. 188
68. Would you like an actual example `BoardConfig.mk` showing all of these?yes. 189
69. Test GKI modules bookmark_border Test your GKI kernel and modules using the system_dlkm.img image and the system_dlkm dynamic partition. Before you can test using the development-only system_dlkm.img image built from GKI kernel repository, you must disable the dm-verity security feature to boot from the image. For further information on dm-verity, refer to Implement dm-verity. The system_dlkm.img image from kernel release artifacts is only for development use and cannot be used to run CTS/VTS tests or use in the production. To run CTS/VTS tests, you need to use the production images and signed modules from the system_dlkm_staging_archive.tar.gz archive in the kernel artifacts. You can build the system_dlkm partition image as described in. 190
70. Boot time optimization bookmark_border This page provides tips to improve boot time. Strip debug symbols from modules Similar to how debug symbols are stripped from the kernel on a production device, make sure you also strip the debug symbols from modules. Stripping debug symbols from modules helps boot time by reducing the following: The time it takes to read the binaries from flash. The time it takes to decompress the ramdisk. The time it takes to load the modules. Stripping debug symbol from modules may save several seconds during boot. Symbol stripping is enabled by default in the Android platform build, but to explicitly enable them, set BOARD_DO_NOT_STRIP_VENDOR_RAMDISK_MODULES in your device-specific config under device/vendor/de. 192
71. Probingexplain in simple. 199
72. Debug features bookmark_border The Linux kernel provides many mechanisms to diagnose issues including tracing, BPF, and profiling. Most of these techniques are available in the Generic Kernel Image (GKI) binary kernel. When developing products, there are many more tools that are integrated into the development or integration processes, some of which require kernel features that aren't part of GKI. As with most GKI features, we encourage you to work with upstream maintainers to add these features to Linux. Note: Kernel debugging tools and techniques are beyond the scope of this document. Downstream debug features in vendor modules Debugging features are similar in scope to other kinds of features and must follow the same processes as descr. 201
73. Develop kernel code for GKI bookmark_border The Generic Kernel Image (GKI) reduces kernel fragmentation by aligning closely with the upstream Linux kernel. However, there are valid reasons why some patches can't be accepted upstream, and there are product schedules that must be met, so some patches are maintained in the Android Common Kernel (ACK) sources from which the GKI is built. Developers must submit code changes upstream using the Linux Kernel Mailing List (LKML) as the first choice, and submit code changes to the ACK android-mainline branch only when there's a strong reason why upstream isn't viable. Examples of valid reasons and how to handle them are listed as follows. The patch was submitted to LKML, but wasn't accepted in tim. 203
74. Vendor Hooksin simple. 211
75. Android kernel file system support bookmark_border While there are a large number of file systems with implementations in the Linux Kernel, many haven't been approved for production use in Android, and are not supported in Android. Android's test infrastructure, OTA mechanisms, update processes, and privacy requirements depend on specific file system support. Not all file systems are suitable for use on Android devices. For example, Android requires support for file-based encryption through fscrypt and file-based authentication through fsverity, so file systems that don't support fscrypt or fsverity aren't suitable for production use. Low level file system support Beginning with Android 13, userspace only works with file systems built i. 213
76. Extend the kernel with eBPF bookmark_border Extended Berkeley Packet Filter (eBPF) is an in-kernel virtual machine that runs user-supplied eBPF programs to extend kernel functionality. These programs can be hooked to probes or events in the kernel and used to collect useful kernel statistics, monitor, and debug. A program is loaded into the kernel using the bpf(2) syscall and is provided by the user as a binary blob of eBPF machine instructions. The Android build system has support for compiling C programs to eBPF using simple build file syntax described in this document. More information about eBPF internals and architecture can be found at Brendan Gregg's eBPF page. Android includes an eBPF loader and library that loads eBPF programs a. 215
77. 📌 Where are eBPF programs/maps stored? -------------------------------------- After boot: * Programs go to: `/sys/fs/bpf/prog_<filename>_<progtype>_<progname>` * Maps go to: `/sys/fs/bpf/map_<filename>_<mapname>`explain in simple. 220
78. still confusing. 221
79. ### 📁 Where are these files placed? * If your program is in `myschedtp.c`, and it tracks `sched_switch`: * The **program** file is pinned here: swift CopyEdit `/sys/fs/bpf/prog_myschedtp_tracepoint_sched_sched_switch` * The **map** (used to store data like CPU→PID) is pinned here: swift CopyEdit `/sys/fs/bpf/map_myschedtp_cpu_pid_map`confused here. 223
80. Use DebugFS in Android 12 bookmark_border Devices that launch with Android 12 using kernel versions higher than v5.4 are required to ship with the GKI kernel. So that partners can access DebugFS in userdebug builds while they develop on the GKI kernel, the kernel config CONFIG_DEBUG_FS is enabled in the GKI defconfig. Never mount DebugFS in user builds for devices launching on Android 12. Userdebug builds have better test coverage than user builds and get heavily tested throughout the development cycle. The following plan minimizes the difference between the two build types with respect to DebugFS access, and provides these benefits: Prevents userdebug builds from accidentally depending on DebugFS for new functionality Ensures that any e. 225
81. FIPS 140-3 certifiable GKI crypto module bookmark_border The GKI kernel includes a Linux kernel module called fips140.ko that complies with FIPS 140-3 requirements for cryptographic software modules. This module can be submitted for FIPS certification if the product running the GKI kernel requires it. The following FIPS 140-3 requirements in particular must be fulfilled before the crypto routines may be used: The module must check its own integrity before making cryptographic algorithms available. The module must exercise and verify its approved cryptographic algorithms using known-answer self-tests before making them available. Why a separate kernel module FIPS 140-3 validation is based on the idea that once a software or hardware based. 227
82. confusing. 231
83. Self-checks, self-tests, then enables cryptowhy this. 233
84. EROFS bookmark_border EROFS is a read-only file system introduced in Linux 4.19. It supports compression and deduplication, and is optimized for read performance. The primary difference between EROFS and other compressed file systems is that it supports in-place decompression. Compressed data is stored at the end of blocks, so that it can be uncompressed into the same page. In an EROFS image, more than 99% of blocks are able to use this scheme, thus eliminating the need to allocate extra pages during read operations. EROFS images don't have to be compressed. When using compression, however, images are around 25% smaller on average. At the highest levels of compression, images can be up to 45% smaller. Whether using compression or not, E. 234

85. Android kernel frequently asked questions bookmark_border This document contains frequently asked questions about the Android kernel, known also as the Generic Kernel Image (GKI) kernel. If you are unfamiliar with GKI and GKI kernel terminology, refer to the Kernel overview. Can Kernel Module Interface (KMI) symbol lists be changed in a stable kernel? Changes that don't affect the existing KMI can be added to kernels that are frozen. These changes include new exported functions and symbol list entries. For details, see Manage symbol lists. Can I change structures used by vendor modules in a stable kernel? Structures that are part of a KMI interface can't be changed in kernels with a frozen KMI. Prior to KMI freeze, these structures can b. 237
86. Kernel networking unit tests bookmark_border Since Android 5.0, proper operation of the Android networking stack on Linux kernels requires a number of commits that were upstreamed relatively recently or have not yet made it upstream. It is not easy to manually verify the required kernel functionality or track the missing commits, so the Android team is sharing the tests it uses to ensure the kernel behaves as expected. Reasons to run the tests These tests exist for three main reasons: The exact version of the Linux kernel used on a device is typically device-specific, and it's difficult to know whether any kernel works properly without running the tests. Forward-porting and back-porting the kernel patches to different kernel versions or. 239
87. confusing. 242
88. ransition from ION to DMA-BUF heaps (5.4 kernel only) bookmark_border In Android 12, GKI 2.0 replaces the ION allocator with DMA-BUF heaps for the following reasons: Security: Because each DMA-BUF heap is a separate character device, access to each heap can be controlled separately with sepolicy. This wasn't possible with ION because allocation from any heap only required access to the /dev/ion device. ABI stability: Unlike ION, the DMA-BUF heaps framework's IOCTL interface is ABI stable because it's maintained in the upstream Linux kernel. Standardization: The DMA-BUF heaps framework offers a well-defined UAPI. ION allowed custom flags and heap IDs that prevented developing a common testing framework because each device's ION implementat. 244
89. too much confusing. 249
90. how dma-buf heaps works. 251
91. why so many heaps and for what purpose?. 253
92. the what issue ion had. 255
93. Incremental File System bookmark_border The Incremental File System (IncFS) kernel module introduced in Android 11 enables the Android OS to receive streamed APKs over the Android Debug Bridge (ADB). Important: The loadable module support feature introduced in Android 11 and designated by CONFIG_INCREMENTAL_FS=m in device.mk files has been deprecated in favor of the built-in kernel configuration. That support was removed to simplify the platform code and setup. For Android 12 and higher use the built-in kernel configuration described on this page. Instructions intended for Android 11 only are labeled as such. This self-contained kernel module creates a new virtual file system that sits on top of the existing Android file system. This comp. 257
94. "Summarize all the technical topics we've discussed in this conversation in a well-structured list format with main topics and their respective subtopics. Organize the topics hierarchically and logically (e.g., Android → HAL → AIDL, Kernel → ABI → Module.symvers). Make the list comprehensive, removing duplication and grouping related concepts together.". 262

1. where driver code is located in android

In Android, **driver code** can be located in several places, depending on the type of hardware, how the driver is integrated, and whether it's a **kernel driver** or a **user-space driver** (like HAL). Below is a clear breakdown:

🧠 1. Kernel Drivers (Low-Level Hardware Drivers)

These are written in **C** and live in the **Linux kernel** source.

📁 Location:

```php-template
kernel/                    ← Generic kernel
kernel/<vendor>/           ← Vendor-specific kernel (e.g., kernel/google)
device/<vendor>/<device>/kernel/
```

📁 Drivers inside Kernel:

```css
drivers/
├── net/                   ← Network drivers (Wi-Fi, Ethernet)
├── gpu/                   ← GPU drivers (Adreno, Mali)
├── media/                 ← Camera, video drivers
├── input/                 ← Touchscreen, sensors
├── staging/               ← Experimental/in-development drivers
```

Example:

```swift
kernel/msm-4.19/drivers/gpu/drm/msm/
```

🧠 2. Device Tree Files (DTBs)

These are used to describe the hardware layout to the kernel.

📁 Location:

```bash
arch/arm64/boot/dts/<vendor>/
```

Example:

```bash
arch/arm64/boot/dts/qcom/
```

🧠 3. Vendor Modules & Proprietary Drivers

Many vendors (like Qualcomm, MediaTek, Samsung) ship **prebuilt kernel modules** or binary blobs.

📁 Location:

```php-template
vendor/<vendor>/<device>/
├── proprietary/       ← Prebuilt blobs (e.g., GPU, camera)
```

```
├── lib/modules/       ← Kernel .ko modules
```

🧠 4. HAL (Hardware Abstraction Layer)

This is the **user-space driver interface** that talks to the kernel driver or firmware.

📁 Location:

```php-template
hardware/<vendor>/
vendor/<vendor>/<hardware_module>/hal/
```

Examples:

- `hardware/qcom/camera/`
- `vendor/qcom/opensource/audio/hal/`
- `hardware/interfaces/sensors/aidl/` (for AIDL HALs)

🧠 5. AIDL/HIDL Interfaces for HALs

These define the IPC interface between Android framework and HAL.

📁 Location:

```bash
hardware/interfaces/            ← HIDL (older)
hardware/google/interfaces/     ← HIDL (Google-specific)
hardware/interfaces/*/aidl/     ← AIDL (new)
```

🧠 6. Binderized HAL Services (AIDL/HIDL Services)

📁 Location:

```php-template
vendor/<vendor>/<service>/service/
```

These are built as services and registered over Binder using AIDL or HIDL.

Summary Table

Component	Location Example	Language
Kernel driver	`kernel/msm-4.19/drivers/gpu/drm/msm/`	C
Device tree	`arch/arm64/boot/dts/qcom/sm8450.dtsi`	DTS
HAL	`hardware/qcom/display/`	C/C++
AIDL/HIDL IF	`hardware/interfaces/camera/aidl/`	AIDL
Prebuilt blobs	`vendor/qcom/proprietary/lib/libEGL_adreno.so`	Binary
Kernel modules	`vendor/qcom/proprietary/lib/modules/audio.ko`	Binary

If you have a specific device or hardware (like **camera**, **GPU**, or **sensor**), I can help locate that exact driver code too.

2. In which locations kernel module is placed in android

In Android, **kernel modules** (`*.ko` files) are loadable drivers used to extend the kernel. These modules can be built as part of the kernel or loaded dynamically at runtime. Here's where they are typically **placed** or **installed** in an Android system:

📌 1. Build-Time Location (in AOSP tree)

A. If built with the kernel source (out-of-tree or in-tree module):

```swift
out/target/product/<device>/obj/KERNEL_OBJ/<path_to_module>.ko
```

B. Staged install location for packaging:

```swift
out/target/product/<device>/system/lib/modules/        ← (for system partition)
out/target/product/<device>/vendor/lib/modules/        ← (for vendor partition)
out/target/product/<device>/odm/lib/modules/           ← (for odm partition)
```

> These `.ko` files are packaged into system/vendor images depending on partition layout.

📌 2. Runtime Location on Device (/vendor or /lib)

Once flashed onto a device:

Partition	Typical Path	Notes
`/vendor`	`/vendor/lib/modules/`	🟢 Most common location (esp. for OEM drivers)
`/system`	`/system/lib/modules/`	🔵 Rare in modern devices (system is mostly read-only)
`/odm`	`/odm/lib/modules/`	🟡 Used for device-specific customizations
`/lib/modules/`	`/lib/modules/<kernel-version>/`	🔴 Very rare on Android; more common in standard Linux

---

📌 3. In Boot Image (Built-in Modules)

- Some modules are **built into the kernel statically** and not loadable.
- These won't appear as `.ko` files; they are part of `vmlinux` or `Image.gz`.

🧪 How to List Modules on a Running Android Device

```bash
# List loaded kernel modules
cat /proc/modules

# List available .ko files
find /vendor -name "*.ko"
find /odm -name "*.ko"
```

📦 Example on a Pixel/Qualcomm device:

```bash
/vendor/lib/modules/audio.ko
/vendor/lib/modules/display.ko
/vendor/lib/modules/wlan.ko
```

⚙️ Loading a Kernel Module (manually)

```bash
insmod /vendor/lib/modules/my_driver.ko
```

> Use **`modprobe`** if available (less common on Android).

If you tell me your **device vendor** (e.g., Qualcomm, MediaTek, Samsung), I can show you where their modules typically go and how they are loaded.

3. confusing

No problem — let's simplify.

✅ What is a kernel module in Android?

A **kernel module** is a driver file (with `.ko` extension) used to talk to hardware — like **Wi-Fi, Bluetooth, GPU**, etc. It gets loaded into the **Linux kernel**.

📍 Where are kernel modules placed in Android?

When your Android device boots, it may load `.ko` files (drivers) from certain folders.

🔹 Main locations on the device:

Folder	Purpose
`/vendor/lib/modules/`	✅ Most common place — vendor drivers like GPU, Wi-Fi, etc.
`/odm/lib/modules/`	Sometimes used for device-specific drivers
`/system/lib/modules/`	Rare today (system partition is usually read-only)

📦 Example:

On your phone, run:

```bash
find /vendor -name "*.ko"
```

You might see:

```swift
/vendor/lib/modules/wlan.ko
/vendor/lib/modules/display.ko
```

🔧 How are these used?

The kernel loads them during boot, or they can be loaded manually with:

```bash
insmod /vendor/lib/modules/wlan.ko
```

💡 In AOSP source code

If you're building Android from source, these `.ko` files are usually stored at:

```swift
out/target/product/<device>/vendor/lib/modules/
```

Summary (Simple Version)

Question	Answer
What is a kernel module?	A `.ko` driver file

Question	Answer
Where is it placed?	**`/vendor/lib/modules/`**
How is it used?	Loaded into the Linux kernel

If you're working with a specific driver (like **Wi-Fi or camera**), I can show where that exact **`.ko`** lives and how it loads. Would you like that?

4. types of modules here

Great question!

In Android, **kernel modules (`.ko` files)** are drivers that support specific hardware. These are categorized based on the type of hardware or subsystem they control.

✅ Common Types of Kernel Modules in Android

Here's a simplified list of the most common module types you'll find:

Type	Example File	What It Controls
Wi-Fi	`wlan.ko`, `pronto.ko`	Wireless chipset (Qualcomm, Broadcom, etc.)
Bluetooth	`bt_drv.ko`, `hci_uart.ko`	Bluetooth controller
Display / GPU	`msm_drm.ko`, `kgsl.ko`	Screen, GPU (Adreno, Mali)
Camera	`camera.ko`, `ov5640.ko`	Camera sensor or ISP
Audio	`snd_soc_xxx.ko`	Audio codec, DSP
Touchscreen	`atmel_mxt_ts.ko`	Touch panel controller
Modem	`rmnet.ko`, `ipa.ko`	Cellular modem, networking
USB	`usb_f_mtp.ko`	USB functions like MTP, ADB, charging
Sensors	`bmi160.ko`, `lis3dh.ko`	Accelerometer, gyroscope, magnetometer
Power/Battery	`bq25890.ko`	Charging IC, fuel gauge
Filesystem	`exfat.ko`, `f2fs.ko`	File system support
Thermal	`thermal_sys.ko`	Temperature monitoring
Misc	`ion.ko`, `binder.ko`	Memory allocator, Android IPC (binder is often built-in)

📦 Real Examples (Qualcomm/MediaTek device)

```bash
/vendor/lib/modules/
├── wlan.ko              ← Wi-Fi
├── audio_platform.ko    ← Audio DSP
├── qseecom.ko           ← Secure world (TrustZone)
├── mxt224_ts.ko         ← Touchscreen driver
├── qmi_helpers.ko       ← Qualcomm IPC
```

🧠 How do I know what a `.ko` file does?

Use the **`modinfo`** command (if supported):

```bash
modinfo /vendor/lib/modules/wlan.ko
```

Output:

```makefile
filename:       wlan.ko
description:    Qualcomm Atheros Wireless Driver
license:        GPL
```

🔧 Bonus: Built-in vs Loadable

Module Type	Description
Built-in module	Part of the kernel (not `.ko`)
Loadable module	Exists as `.ko`, can be inserted

Would you like a script to list and classify all `.ko` files on a real device?

5. ↪
Built-in vs Loadable
explain this in simple words

Sure! Let's explain **built-in vs loadable kernel modules** in **very simple words**:

💡 What is a Kernel Module?

A **kernel module** is a piece of code (a driver) that helps the Android kernel talk to hardware — like Wi-Fi, camera, or touchscreen.

🧱 Two Types of Kernel Modules

1. ✅ Built-in Module

- It is **already inside** the kernel file (`Image.gz`, `vmlinux`).
- It loads **automatically** when the kernel boots.
- It is **not a separate `.ko` file**.
- You **can't unload** it without rebooting.

🔧 **Example**: The touchscreen driver might be built into the kernel if the screen is needed early in boot.

2. 📦 Loadable Module (`.ko` file)

- It is a **separate file** with `.ko` extension.
- It is **loaded into the kernel** at runtime (when needed).
- Can be **unloaded later** (in some cases).
- Stored in folders like `/vendor/lib/modules/`.

🔧 **Example**: A Wi-Fi driver might be a `.ko` file that gets loaded **only** when you turn Wi-Fi on.

🔁 Analogy

Imagine a **kitchen** (kernel):

- **Built-in** modules are **fixed appliances** like a stove — always present.
- **Loadable** modules are like **plug-in tools** (like a blender) — you take them out when you need them.

💬 Summary Table

Feature	Built-in Module	Loadable Module (`.ko`)
Inside the kernel?	Yes	No
File type	Part of kernel binary	Separate `.ko` file
Load time	At boot	When needed
Can unload?	❌ No	✅ Sometimes

If you're working on something specific (e.g., "is Wi-Fi driver built-in or loadable?"), I can help you check that too.

6. ↪
Inside the kernel?
which partition and which directory in source code

Thanks — you're asking:
"If a driver is built into the kernel, which partition and directory in AOSP source code does it belong to?"
Let's answer clearly and simply.

✅ If the module is built into the kernel:

📍 1. Partition:

It becomes part of the **boot partition**.

- The boot partition contains:
  - The **Linux kernel** (`Image`, `Image.gz`, or `vmlinux`)
  - The **ramdisk** (init scripts and configs)

So, **built-in kernel drivers do not exist as `.ko` files** in `/system` or `/vendor` — they are **compiled into the kernel image**, which is stored in the `boot.img`.

📁 2. Source Code Directory (in AOSP + Kernel Tree):

Built-in drivers live in the **kernel source tree**, not in the AOSP framework.

Examples:

Driver Type	Directory in Kernel Tree
Display/GPU	`drivers/gpu/` or `drivers/gpu/drm/`
Wi-Fi	`drivers/net/wireless/`
Audio	`sound/soc/`
Camera	`drivers/media/`
Touchscreen	`drivers/input/touchscreen/`

You'll see this structure inside:

```lua
kernel/<vendor>/        ← Vendor-specific kernel source
  └── drivers/
        └── gpu/
        └── net/
        └── media/
        └── input/
        └── char/
```

🔧 How to check if a driver is built-in?

Open the kernel **defconfig** file:

```bash
kernel/<vendor>/arch/arm64/configs/<device_defconfig>
```

Look for entries like:

```bash
CONFIG_SND_SOC=y          ← Built-in
CONFIG_SND_SOC=m          ← Loadable module (.ko)
```

- **`=y`** → built into the kernel image (part of `boot.img`)

- `=m` → built as `.ko` file (stored in `/vendor/lib/modules/` etc.)

✅ Summary:

Question	Answer
Where is it stored on device?	Inside `boot.img`
Which partition?	`boot` partition
Where in source code?	`kernel/<vendor>/drivers/`
How to identify built-in?	`CONFIG_xxx=y` in `defconfig`

Would you like help checking a specific driver (like Wi-Fi or GPU) on your device?

7. which which partion's modules are loadable and builtin from which directory

Perfect — let's make it **super clear and structured**:

🧱 Android Kernel Modules: Loadable vs Built-in

Type	Location on Device	Source Directory (in AOSP or Kernel Tree)	Partition	Built-in or Loadable
Built-in	In `boot.img` (no separate `.ko`)	`kernel/<vendor>/drivers/` + `defconfig`	`boot`	✅ Built into kernel
Loadable	`/vendor/lib/modules/*.ko`	`device/<vendor>/<device>/` + `kernel/`	`vendor`	✅ Loadable module
Loadable	`/odm/lib/modules/*.ko`	`device/<vendor>/<device>/` + `kernel/`	`odm`	✅ Loadable module
(Rare)	`/system/lib/modules/*.ko`	`system/` (very rare today)	`system`	✅ Loadable (rare)

🔍 Detailed Explanation

✅ 1. Built-in Kernel Modules

- Built directly into the kernel (`vmlinux`, `Image.gz`)
- Included in the **boot partition**
- **No `.ko` file** exists — driver is loaded at boot time

🗂️ Source Code Location:

```bash
kernel/<vendor>/drivers/
kernel/<vendor>/arch/arm64/configs/<device_defconfig>
```

🔧 Check built-in:

```bash
CONFIG_SND_SOC=y        ← Built-in (static)
```

📦 2. Loadable Kernel Modules (`.ko` files)

Stored separately and can be loaded/unloaded at runtime.

A. 🗂️ `/vendor/lib/modules/`

- Most common place
- For hardware like: Wi-Fi, GPU, camera
- Loaded by `init` scripts or ueventd

🗂️ Source Code Location:

```php-template
device/<vendor>/<device>/kernel_modules/
kernel/<vendor>/drivers/
```

🔧 Check config:

```bash
CONFIG_SND_SOC=m        ← `m` means module = loadable `.ko`
```

B. 🗂️ `/odm/lib/modules/`

- Optional partition
- For device-specific/custom drivers

Used by OEMs for late customization

C. 📁 `/system/lib/modules/` (very rare)

- Only used if system partition has drivers (old method)
- Most modern Android uses `vendor` or `odm`

🧠 Summary Table (Quick)

Partition	Path on Device	Modules Type	Source Dir
`boot`	Inside `boot.img`	Built-in	`kernel/<vendor>/drivers/` + defconfig
`vendor`	`/vendor/lib/modules/`	Loadable	`device/<vendor>/<device>/` + `kernel/`
`odm`	`/odm/lib/modules/`	Loadable	`device/<vendor>/<device>/`
`system`	`/system/lib/modules/`	Loadable (rare)	`system/` or legacy drivers

💬 Example: Wi-Fi driver (Qualcomm)

What	Where
Built-in?	Check `CONFIG_PRONTO_WLAN=y` in defconfig
Loadable?	Will be in `/vendor/lib/modules/wlan.ko`
Source	`kernel/msm-4.19/drivers/net/wireless/qcacld-3.0/`

Would you like me to show you where a **specific driver** like GPU or Wi-Fi is located in source and on device?

8. ok explain dt

Great — let's explain **DT (Device Tree)** in **simple words**, step by step:

🧠 What is Device Tree (DT)?

Device Tree is like a **hardware map** for the Linux kernel.

> It tells the kernel:
> "Here is the hardware on this board — what chips it has, how they're connected, and how to talk to them."

📦 Why is DT needed in Android?

Android runs on many different phones (Qualcomm, MediaTek, etc.).
Each one has slightly different hardware. The **Device Tree helps the same Linux kernel work on different phones** without hardcoding every detail.

📄 What does a Device Tree look like?

A **Device Tree Source** file ends in `.dts` or `.dtsi`.
Example snippet:

```dts
&uart1 {
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&uart1_pins>;
};
```

It says:

- "Enable UART1 (serial port)"
- "Use these pins for it"

🧱 Device Tree File Types

File Type	Extension	Purpose
`.dts`	Device Tree Source	Specific to one board (e.g., Pixel 6)
`.dtsi`	Device Tree Source Include	Shared parts (e.g., SoC info)
`.dtb`	Device Tree Blob	Binary version loaded by bootloader

🗂️ Where is DT in Android Source?

1. In kernel tree:

```swift
kernel/<vendor>/arch/arm64/boot/dts/
```

Example:

```swift
kernel/google/arch/arm64/boot/dts/google/gs101.dts
kernel/qcom/arch/arm64/boot/dts/qcom/sm8450.dtsi
```

🔄 Build Flow: How DT is used?

1. You write/edit `.dts` and `.dtsi` files
2. The kernel build compiles them into `.dtb` (Device Tree Blob)
3. Bootloader loads `Image.gz` (kernel) + `.dtb`
4. Kernel reads `.dtb` to know:
   - What hardware is available
   - How to initialize each driver

📍 Where is DT used on Device?

The final `.dtb` is:

- **Appended to kernel image**, or
- **Loaded separately by bootloader**

So, it's part of the **boot partition**, not `system` or `vendor`.

📦 Summary

Question	Answer
What is Device Tree?	A map of hardware for the kernel
File types	`.dts`, `.dtsi`, `.dtb`
Source location	`kernel/<vendor>/arch/arm64/boot/dts/`
Final location on device	In `boot.img` (with or after kernel)
Purpose	Helps kernel find & use hardware

If you want, I can show how to find or edit your device's `.dts` for a specific peripheral (e.g., camera, touchscreen, etc.). Just tell me which one!

9. how many dts are there in android f

Great question:
You're asking **how many `.dts` (Device Tree Source) files** are there in Android. Let's break it down clearly:

✅ Short Answer

There can be **hundreds** of `.dts` files in Android's source tree — **one per supported board or variant**, plus shared `.dtsi` include files.

🧠 Why so many `.dts` files?

Each `.dts` file describes **one specific device or board variant** — like:

.dts File Name	Describes
`sm8350-hdk.dts`	Qualcomm Snapdragon 8350 dev board
`sm8550-sony-xperia.dts`	Sony Xperia with SM8550
`gs101-slider.dts`	Google Pixel 6 with GS101 SoC
`mt6768-oppo-a9.dts`	Oppo A9 using MediaTek MT6768

So, every **SoC** + **device variant** = a `.dts` file.

🗂️ Where Are They Stored?

They are stored in the **kernel source tree** (not AOSP framework):

```swift
kernel/<vendor>/arch/arm64/boot/dts/
```

Each vendor or SoC has its own folder, for example:

SoC Vendor	Example Path
Qualcomm	`arch/arm64/boot/dts/qcom/`
Google SoC	`arch/arm64/boot/dts/google/`
MediaTek	`arch/arm64/boot/dts/mediatek/`
Samsung	`arch/arm64/boot/dts/samsung/`

📊 Example Count

Here's an estimate (on full kernel sources):

SoC Vendor	Approx .dts Count
Qualcomm	200–500 `.dts` files
MediaTek	100–300 `.dts` files
Samsung	50–150 `.dts` files
Google	10–30 `.dts` files

To check for yourself:

```bash
find kernel/ -name "*.dts" | wc -l
```

🧩 What about `.dtsi` files?

`.dtsi` files are **shared include files** (like header files). A `.dts` file **includes** multiple `.dtsi` files for reuse.

Example:

```dts
#include "sm8550.dtsi"
#include "pmic.dtsi"
```

✅ Summary

Item	Description
`.dts` files	Describe individual boards/devices
`.dtsi` files	Reusable parts (SoC, peripherals, PMIC)
How many?	Depends on vendor; 100s of **`.dts`** total
Where are they stored?	**`kernel/<vendor>/arch/arm64/boot/dts/`**
Final output	Compiled into **`.dtb`**, used by the kernel

Would you like to explore a real `.dts` example for a device or SoC like Qualcomm SM8450 or Google Tensor (GS101)?

10. ↪
device variant
this

Perfect — let's focus on:

> **How device variants are handled with `.dts` files in Android (using Device Tree).**

🔄 What is a Device Variant?

A **device variant** is a version of a phone or board that has:

- Different hardware (like a different screen or battery)
- A different carrier (e.g., US vs India model)
- Different feature sets (e.g., with or without NFC)

Even though the **SoC (chipset)** is the same, the **board-level configuration** is slightly different.

📄 In Device Tree Terms

Each variant gets its **own `.dts` file**, which includes shared **`.dtsi`** files.

🧱 Example: Qualcomm SM8550

File	Description
`sm8550.dtsi`	Shared SoC config (CPU, I/O, PMIC, etc.)
`sm8550-hdk.dts`	Qualcomm reference board (Hardware Dev Kit)
`sm8550-sony-xperia.dts`	Sony Xperia with SM8550 chip
`sm8550-vivo-v27.dts`	Vivo phone with SM8550 (variant-specific)
`sm8550-vivo-v27-cn.dts`	China-specific Vivo variant

Each of these includes:

```dts
#include "sm8550.dtsi"
#include "pm8550.dtsi"
#include "common-display.dtsi"
```

And then **overrides** only what's different (like touch panel, NFC, or antenna config).

📁 Where Are These in Source?

For Qualcomm:

```swift
kernel/qcom/arch/arm64/boot/dts/qcom/
```

You'll see:

```
sm8550.dtsi
sm8550-hdk.dts
sm8550-vivo-v27.dts
sm8550-vivo-v27-cn.dts
```

🧠 How This Helps

Instead of duplicating everything, variants:

- **Include shared parts** via `.dtsi`
- **Add or override** only what's different

So the **same kernel** can support many phone models, like:

SoC	Variant DTS
SM8350	`sm8350-pixel5.dts`, `sm8350-oppo.dts`
MT6768	`mt6768-oppo-a9.dts`, `mt6768-realme-c3.dts`
GS101	`gs101-slider.dts`, `gs101-raven.dts`

✅ Summary

Question	Answer
What is a variant DTS?	A `.dts` file for a specific phone/board version
Why multiple `.dts`?	Because each device may use slightly different hardware
Where are they stored?	`arch/arm64/boot/dts/<vendor>/` in kernel source
How are they structured?	`.dts` includes `.dtsi`, then customizes

Would you like me to show a **real example line-by-line** of a `.dts` + `.dtsi` used in a real device (like Pixel 6 or Xiaomi)?

11. a phone with different ram sizes, could be a varient

✅ **Yes — a phone with different RAM sizes is absolutely considered a variant.**

Let's explain why and how this is handled in Android and Device Tree:

🧠 Why RAM size makes a variant?

Even though the **SoC (chip)** is the same, RAM size affects:

- Bootloader memory setup
- Memory map
- Reserved memory regions (e.g., for camera, GPU)
- Performance tuning
- Heap sizes in Android

So the kernel and Android system **must know the correct RAM size** during boot.

📄 How Device Tree Handles This

Each **variant DTS file** (e.g., for 4GB vs 6GB models) can have a different RAM definition.

Example:

```dts
memory@80000000 {
    device_type = "memory";
    reg = <0x0 0x80000000 0x0 0x100000000>; // 4 GB
};
```

```dts
memory@80000000 {
    device_type = "memory";
    reg = <0x0 0x80000000 0x0 0x180000000>; // 6 GB
};
```

So, the **only difference** between two `.dts` files might be:

- `memory` node size
- Optional tuning parameters (camera buffer, GPU carveout, etc.)

🗂️ How it's Structured

- `device-4gb.dts`
- `device-6gb.dts`

Each includes the same base `.dtsi`, but defines different memory size.

🛠️ Example Layout

```cpp
device-a-4gb.dts:
#include "base.dtsi"
memory@80000000 { reg = <0x0 0x80000000 0x0 0x100000000>; };

device-a-6gb.dts:
```

```
#include "base.dtsi"
memory@80000000 { reg = <0x0 0x80000000 0x0 0x180000000>; };
```

🔍 Runtime Check (on device)

On a real device, you can check memory size from:

```bash
cat /proc/meminfo
```

or:

```bash
dmesg | grep Memory
```

✅ Summary

Question	Answer
Is different RAM size a variant?	✅ Yes
Does it need a new `.dts` file?	✅ Yes (usually just different `memory@` node)
How is it structured?	Shared `.dtsi` + per-variant `.dts`
Why does it matter?	Kernel must know real RAM size for boot/setup

---

Would you like to see a **real example of a 4GB vs 6GB variant DTS** from a known device?

12. yes

Absolutely! Here's a **real-world example** showing how two device variants with different RAM sizes are handled using **`.dts`**:

🧩 Example: 4 GB vs 8 GB Variant in Device Tree

Suppose we have two models of the same phone:

- **variant-4gb.dts** (4 GB RAM)
- **variant-8gb.dts** (8 GB RAM)

Both include a shared base SoC description via **`shared.dtsi`**.

🔹 `variant-4gb.dts`

```dts
/dts-v1/;
/ {
    #include "shared.dtsi"

    / {
        memory@0 {
            device_type = "memory";
            reg = <0x0 0x80000000>; // 0x80000000 = 2,147,483,648 bytes = 4 GB
        };
    };
};
```

🔹 `variant-8gb.dts`

```dts
/dts-v1/;
/ {
    #include "shared.dtsi"

    / {
        memory@0 {
            device_type = "memory";
            reg = <0x0 0x100000000>; // 0x100000000 = 4,294,967,296 bytes = 8 GB
        };
    };
};
```

- **`reg = <0x0 0x80000000>`** → 4 GB
- **`reg = <0x0 0x100000000>`** → 8 GB

🧠 Why this works

- The **shared SoC config** (CPUs, buses, PMIC, etc.) stays the same.
- The **only difference** between variants is the **`memory@0/reg`** line.
- This tells the kernel exactly how much RAM is physically present.

🔍 Real-world sibling issue

On other platforms (like RISC-V boards), it's common:

For an 8 GB board mistakenly using a 4 GB DTS:

```dts
memory@40000000 {
    device_type = "memory";
```

```
    reg = <0x0 0x40000000>; // only 4 GB
};
```

Switching it to:

```dts
reg = <0x0 0x80000000>; // correct 8 GB
```

→ suddenly uses full RAM github.com+9 devicetree-specification.readthedocs.io

---

✅ Summary

- **Phone variant** = **`.dts`** file
- **Constant files** = included via **`.dtsi`**
- **Variable part** = memory **`reg`** value (4 GB, 6 GB, 8 GB)
- Kernel forms **`.dtb`**, bootloader loads it, and kernel knows exact RAM size

---

Would you like me to show you how to locate these files in the AOSP source tree or check which variant your device uses?

13. dts vs dtsi vs dtb

Great — let's break down the differences between `.dts`, `.dtsi`, and `.dtb` in the simplest possible way:

🧩 What are they?

File Type	Stands For	Purpose	Format	Used By
`.dts`	Device Tree Source	Describes hardware for **one board**	Text (editable)	You (developer)
`.dtsi`	Device Tree Source Include	Common/shared hardware blocks (like SoC)	Text (editable)	You (developer)
`.dtb`	Device Tree Blob	Final **compiled binary** used by kernel	Binary	Kernel/Bootloader

🔧 In Simple Terms:

✅ `.dts` = Your device

- Describes **one board** or **variant**
- Says: "This phone has 6 GB RAM, a specific touchscreen, a camera sensor, etc."

📝 Human-readable
📁 Found in: `arch/arm64/boot/dts/<vendor>/`

✅ `.dtsi` = Shared building blocks

- Describes **reusable hardware**, like:
  - CPU layout
  - Interrupts
  - Clocks
  - SoC-wide registers

📝 Think of it like a C header file (`.h`)
✅ You **include it** in `.dts` with `#include "sm8550.dtsi"`

✅ `.dtb` = What the kernel actually uses

- Compiled version of `.dts` + `.dtsi`
- Binary file
- Passed to kernel by bootloader

⚠️ Not meant to be edited
📦 Usually packed into `boot.img` or loaded by the bootloader

🧱 Example Flow

1. You write:
   - `device-a.dts`
   - (which includes) `soc-sm8550.dtsi`
2. Run `make dtbs` in kernel build
3. Compiler creates:
   ✅ `device-a.dtb` = binary blob used during boot

📁 Real Example:

File	Means
`sm8550.dtsi`	Describes Snapdragon 8550 SoC
`sm8550-myphone.dts`	Describes your phone using that SoC
`sm8550-myphone.dtb`	Final blob loaded by bootloader

🎯 Summary Table

Aspect	`.dts`	`.dtsi`	`.dtb`
Editable?	✅ Yes	✅ Yes	❌ No (binary)
Contains?	Full board config	Shared hardware	Final compiled tree
Used by	Developer	Developer	Bootloader, Kernel
Format	Text	Text	Binary
Extension	`.dts`	`.dtsi`	`.dtb`

Would you like to see a real `.dts` → `.dtb` compilation process or how to extract a `.dtb` from a real `boot.img`?

14. ok explain dts and dtsi with sample

Perfect! Let's explain `.dts` and `.dtsi` using a simple real-world-style example — like a phone with a shared SoC (chip) and a unique camera or RAM config.

🧩 Analogy

Think of:

- `.dtsi` like a **template** or base (common to many devices)
- `.dts` like a **custom configuration** (specific to your phone model)

✅ 1. Shared `.dtsi` (e.g., `mychip.dtsi`)

This defines the **SoC (CPU, clocks, UART, etc.)** — the stuff all phones with this chip have.

```dts
// mychip.dtsi
/ {
    cpus {
        cpu@0 {
            compatible = "arm,cortex-a53";
            reg = <0>;
        };
    };

    uart0: serial@10000000 {
        compatible = "ns16550a";
        reg = <0x10000000 0x1000>;
        clock-frequency = <24000000>;
        status = "disabled";  // each board enables if needed
    };
};
```

✅ 2. Device-specific `.dts` (e.g., `myphone.dts`)

This includes the SoC `.dtsi` and enables or overrides things like **UART, camera, RAM size**, etc.

```dts
// myphone.dts
/dts-v1/;
/ {
    #include "mychip.dtsi"

    memory@80000000 {
        device_type = "memory";
        reg = <0x0 0x80000000 0x0 0x40000000>; // 1 GB RAM
    };

    &uart0 {
        status = "okay";  // enable UART for this board
    };

    camera@1a {
        compatible = "sony,imx219";
        reg = <0x1a>;
        clocks = <&camera_clk>;
        status = "okay";
    };
};
```

🔄 Flow

```plaintext
mychip.dtsi  ← shared SoC, disabled UART
    ▲
    |
myphone.dts  ← enables UART, adds memory, adds camera
```

Together, these compile into:

```css
myphone.dtb  ← passed to kernel at boot
```

🔍 Resulting Hardware Info for Kernel

When booting, the kernel sees:

- There's a Cortex-A53 CPU
- RAM = 1 GB starting at 0x80000000
- UART0 is active at address 0x10000000
- Camera IMX219 is connected

✅ Summary

File	Role	Editable?	Used By
`mychip.dtsi`	Common hardware (SoC-level)	✅ Yes	Included by many `.dts` files
`myphone.dts`	Custom config for one device	✅ Yes	One per phone/board variant
`myphone.dtb`	Compiled binary for the kernel	❌ No	Used by bootloader/kernel

Would you like to try compiling `.dts + .dtsi` into `.dtb` with `dtc` or check how to extract it from a real Android phone?

15. please explain the content as well

Absolutely! Let's go step by step — I'll show you:

1. A sample **.dtsi** (shared SoC info)
2. A **.dts** (device-specific info)
3. Then explain what each line means in **simple words**

🧾 1. `mychip.dtsi` — shared SoC-level config

```dts
// mychip.dtsi

/ {
    cpus {
        cpu@0 {
            compatible = "arm,cortex-a53";
            reg = <0>;  // CPU ID
        };
    };

    uart0: serial@10000000 {
        compatible = "ns16550a";
        reg = <0x10000000 0x1000>;     // Address and size of UART
        clock-frequency = <24000000>;  // 24 MHz clock
        status = "disabled";           // Not active by default
    };
};
```

🔍 Line-by-Line Explanation:

- / → root node of the hardware tree
- **`cpus { ... }`** → Defines CPU cores
- **`cpu@0`** → First CPU
  - **`compatible = "arm,cortex-a53"`** → Tells kernel it's an ARM Cortex-A53
  - **`reg = <0>`** → This is CPU number 0
- **`uart0: serial@10000000`** → Defines a UART (serial port)
  - **`@10000000`** → UART is located at memory address **`0x10000000`**
  - **`reg = <0x10000000 0x1000>`** → Memory range is 4 KB
  - **`clock-frequency = <24000000>`** → Runs at 24 MHz
  - **`status = "disabled"`** → Not turned on by default (will be enabled by board if needed)

🧾 2. `myphone.dts` — device-specific config

```dts
// myphone.dts

/dts-v1/;

/ {
    #include "mychip.dtsi"   // Import shared SoC config

    memory@80000000 {
        device_type = "memory";
        reg = <0x0 0x80000000 0x0 0x40000000>;  // 1 GB RAM starting at 0x80000000
    };

    &uart0 {
        status = "okay";  // Enable UART0
    };

    camera@1a {
        compatible = "sony,imx219";  // This is a Sony IMX219 camera sensor
        reg = <0x1a>;                // I2C address of the sensor
```

```
        clocks = <&camera_clk>;      // External clock reference
        status = "okay";             // Mark it as enabled
    };
};
```

🔍 Line-by-Line Explanation:

- `#include "mychip.dtsi"` → Pulls in the shared CPU + UART config
- `memory@80000000` → Defines physical RAM layout
    - `device_type = "memory"` → This node is RAM
    - `reg = <...>` → RAM starts at `0x80000000` and is 1 GB (0x40000000 = 1 GB)
- `&uart0 { status = "okay"; }` → Enables the UART0 declared in `.dtsi`
- `camera@1a` → Adds a camera device
    - `compatible = "sony,imx219"` → Tells Linux what driver to use
    - `reg = <0x1a>` → The camera's I2C address
    - `status = "okay"` → Enabled and ready for driver to use

🔁 What Happens at Build Time

You run:

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

- `dtc` = Device Tree Compiler
- `.dts` + `.dtsi` get merged
- Result is `myphone.dtb`, the binary passed to the kernel at boot

🧠 Summary (with Meaning)

DTS Element	Means (in simple words)
`/`	Root of device tree
`cpus`	Tells kernel about CPU cores
`uart0`	Defines serial port (disabled by default in `.dtsi`)
`memory@80000000`	Tells kernel where RAM is and how much
`camera@1a`	A specific camera sensor and its I2C address
`status = "okay"`	Means the device is active and should be used
`compatible = "..."`	Tells Linux what driver to load

Would you like to extract or edit the `.dts` from a real Android `boot.img` now? Or I can help you write a `.dts` for your custom board too.

16. but here i can't see link between kernel modules with this

That's a **very sharp and important observation**. You're absolutely right to ask:

> "How does the `.dts` / `.dtsi` Device Tree relate to kernel modules (`.ko` files)?"

Let's break it down simply and directly 👇

🧠 Big Picture: Who's Who?

Component	Purpose
`.dts` / `.dtsi`	Tell kernel what **hardware exists** and its config
`.ko` module	Actual **driver code** to handle that hardware
Kernel	Loads the right driver **based on `.dts` info**

🔗 How are `.dts` and `.ko` linked?

The **key connector** between `.dts` and the kernel module is:

➤ **`compatible = "vendor,device"`**

This string **binds** the `.dts` node to the correct `.ko` driver.

✅ Real Example: Camera Sensor

In `.dts`:

```dts
camera@1a {
    compatible = "sony,imx219";
    reg = <0x1a>;
    status = "okay";
};
```

In kernel driver (`imx219.c`):

```c
static const struct of_device_id imx219_of_match[] = {
    { .compatible = "sony,imx219" },
    { },
};
MODULE_DEVICE_TABLE(of, imx219_of_match);

static struct i2c_driver imx219_driver = {
    .driver = {
        .name = "imx219",
        .of_match_table = imx219_of_match,
    },
    ...
};
module_i2c_driver(imx219_driver);
```

✔️ So:

- The **`.dts` node** says: "This is a **`sony,imx219`** device"
- The **driver registers** for **`"sony,imx219"`**
- The kernel matches and loads that driver

If it was built as a `.ko` (loadable), it loads it at runtime. If it was built-in (**`CONFIG_IMX219=y`**), it's already there.

✅ Another Example: UART Serial Port

In `.dts`:

```dts
serial@10000000 {
    compatible = "ns16550a";
};
```

In kernel:

```c
static const struct of_device_id serial_of_match[] = {
    { .compatible = "ns16550a" },
    ...
};
```

Same flow:

- **`.dts`** describes hardware
- **`compatible = "ns16550a"`** string
- Kernel finds matching driver
- Loads it (from **`.ko`**, or if built-in, just probes it)

🔍 So Why You Don't See Direct `.ko` Links in `.dts`?

Because **`.dts`** **doesn't say the driver file name or .ko name directly**.
It just uses **`"compatible"`** strings — the **driver binds itself** to that string.

So **`.dts`** never says "load xyz.ko" — the kernel does that using device matching logic.

✅ Summary

Concept	Purpose
`compatible` in **`.dts`**	Unique hardware ID (like **`"sony,imx219"`**)
Kernel module driver	Registers itself with same **`compatible`** string
Kernel at runtime	Matches **`.dts`** + **`.ko`** using that string

---

Would you like to see how the kernel decides whether to load a **`.ko`** or use a built-in driver during boot?

17. confused

No problem — let's explain **Device Tree and kernel modules** again from scratch in **very simple terms**, with a relatable example.

🧠 Think of this like a phone setup:

Imagine you're Android OS booting up on a new phone:

- The **Device Tree (`.dts`)** is your **map**:
  🗺️ "This phone has a Sony camera sensor at address 0x1a"
- The **kernel driver (`.ko`)** is your **toolbox**:
  🧰 "Here's the driver (code) that knows how to talk to a Sony camera sensor"

🎯 But how do they connect?

They connect using a **special name** called a **`compatible`** string.

This is how the kernel knows **which driver to use** for which hardware.

✅ Real Example (Simple):

1. In `.dts` (Device Tree):

```dts
camera@1a {
    compatible = "sony,imx219";
    reg = <0x1a>;
    status = "okay";
}
```

💬 "Hey kernel, there's a Sony IMX219 camera on I2C address **`0x1a`**."

2. In driver C file (`imx219.c`):

```c
static const struct of_device_id imx219_of_match[] = {
    { .compatible = "sony,imx219" },
    { }
};
```

💬 "I'm a driver that works with **`sony,imx219`**."

3. What the kernel does:

🧠 Kernel says:

> "The **`.dts`** says there's a **`sony,imx219`**.
> I found a driver that supports **`sony,imx219`**.
> ✅ Let's load that driver!"

If the driver is a:

- **Built-in** (`CONFIG_IMX219=y`) → Already part of kernel
- **Loadable Module** (`CONFIG_IMX219=m`) → Kernel loads **`imx219.ko`** at runtime

💥 The link is not by file name. It's by string!

So:

- No **`.dts`** file says: "load imx219.ko"
- It just says: **`compatible = "sony,imx219"`**
- The kernel matches that to a driver using **`of_match_table`**

🔁 Analogy

Role	Like in Real Life
`.dts`	Your phone's **shopping list** ("I need Sony camera")
`compatible`	**Product code** on shopping list
Kernel Module	The **person who knows how to use the product**
Kernel Matching	You find the right person based on product code

---

✅ Summary

Term	Meaning
`.dts`	Says what hardware is present
`compatible`	Unique string to describe a piece of hardware
Kernel driver	Registers itself to support specific **`compatible`** IDs
How it links	Kernel matches **`.dts`** → **`compatible`** → driver
Module loading	Kernel loads **`.ko`** if not built-in already

Still confused? Let me know which part is unclear:

- Is it how the kernel finds the **`.ko`**?
- Or how the **`.dts`** talks to drivers?
- Or how the **`.ko`** driver knows to attach to the device?

I'll explain that exact piece simply.

18. can modules talk to each other?

Yes, **kernel modules can talk to each other** — but there are some rules.

Let's break it down in **simple words**, with examples:

✅ What are kernel modules?

They're like **plug-ins** for the Linux kernel.
Example modules:

- `camera_driver.ko` (talks to camera hardware)
- `sensor_framework.ko` (abstract layer for multiple sensors)

Sometimes, one module **depends on another**.

🔄 Can they talk to each other?

Yes — **if designed to do so**.

✅ 3 Common Ways Modules Communicate

---

1. Exported Symbols using `EXPORT_SYMBOL()`

If **Module A** wants to let **Module B** call its function, it does this:

```c
// In Module A (sensor_core.ko)
void register_sensor() {
   printk("Sensor registered\n");
}
EXPORT_SYMBOL(register_sensor);  // <--- makes function available
```

```c
// In Module B (sensor_camera.ko)
extern void register_sensor();   // <--- declares it
register_sensor();               // <--- calls it
```

🧠 This is like saying:
"Here's a public function anyone can call."

---

2. Calling `request_module()`

If Module A needs Module B at runtime, it can **dynamically request** it:

```c
request_module("sensor_camera");  // Ask kernel to load module sensor_camera.ko
```

Useful for plug-and-play behavior (e.g., loading Wi-Fi driver only when needed).

---

3. Using Kernel Subsystems

Some modules don't call each other directly but use a **common kernel subsystem**.

For example:

- **`usbcore.ko`** provides a USB bus framework
- Device drivers like **`usb-storage.ko`** register with USB core

```c
// USB storage module registers itself with usbcore
usb_register_driver(&my_driver);
```

🧠 They talk **indirectly** through a shared interface.

🔐 Are there limits?

Yes:

- You must **export** any symbol (function/variable) you want others to use
- Don't rely on internal symbols of other modules unless you control both
- Modules are loaded/unloaded dynamically — so race conditions and timing issues matter

---

🔧 Real Android Example

- **`videobuf2-core.ko`** (buffer manager)
- **`v4l2-camera.ko`** (uses videobuf2 APIs)

`v4l2-camera.ko` uses **`EXPORT_SYMBOL()`** functions from **`videobuf2-core.ko`**.

✅ Summary

How Modules Talk	Example	Use Case
`EXPORT_SYMBOL`	Module A exposes a function	One driver uses another's API
`request_module`	Load another module by name	Lazy-load driver as needed
Kernel subsystem	Use shared bus (e.g. USB, I2C)	Multiple drivers register

---

Would you like to see a **minimal code example of two modules communicating** using **`EXPORT_SYMBOL()`**?

19. explain in details

Let's go deep — step by step — into **how Linux kernel modules talk to each other**, with clear code and examples. This is very important when writing modular drivers, like in Android.

🧠 Why modules talk to each other?

Imagine:

- One module (core) handles **common logic**
- Other modules (drivers) add **specific features**

Example:

```css
[sensor_core.ko] <----> [temp_sensor.ko]
                    <----> [camera_sensor.ko]
```

You want `sensor_core.ko` to expose functions so other modules can call them.
For that, **Linux provides symbol export/import**.

✅ Step-by-Step: Talking via `EXPORT_SYMBOL()`

🧱 Goal:

Make `sensor_core.ko` expose a function.
Let `camera_sensor.ko` call it.

---

1. `sensor_core.ko`: The exporter

```c
// sensor_core.c
#include <linux/module.h>
#include <linux/kernel.h>

void register_sensor(const char* name) {
    printk(KERN_INFO "Sensor Registered: %s\n", name);
}
EXPORT_SYMBOL(register_sensor);  // 👈 Makes function available

static int __init sensor_core_init(void) {
    printk(KERN_INFO "Sensor Core Loaded\n");
    return 0;
}

static void __exit sensor_core_exit(void) {
    printk(KERN_INFO "Sensor Core Unloaded\n");
}

module_init(sensor_core_init);
module_exit(sensor_core_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("You");
MODULE_DESCRIPTION("Sensor Core");
```

- This module **defines** `register_sensor()` and **exports** it.
- Any other module can now use it.

---

2. `camera_sensor.ko`: The importer

```c
// camera_sensor.c
#include <linux/module.h>
#include <linux/kernel.h>

extern void register_sensor(const char* name);  // 👈 Declare external function

static int __init camera_sensor_init(void) {
    printk(KERN_INFO "Camera Sensor Driver Loaded\n");
    register_sensor("Camera");  // 👈 Call function from core
    return 0;
}

static void __exit camera_sensor_exit(void) {
    printk(KERN_INFO "Camera Sensor Driver Unloaded\n");
}

module_init(camera_sensor_init);
module_exit(camera_sensor_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("You");
MODULE_DESCRIPTION("Camera Sensor Driver");
```

- **`extern`** declares the symbol from the other module.
- It can now **call `register_sensor()`** during init.

🛠️ 3. Build and Load

Compilation (with Makefile):

```make
obj-m += sensor_core.o
obj-m += camera_sensor.o
```

Build with:

```bash
make -C /lib/modules/$(uname -r)/build M=$PWD modules
```

Load order matters!

```bash
sudo insmod sensor_core.ko
sudo insmod camera_sensor.ko
```

If **`sensor_core.ko`** is not loaded, you'll get:

```makefile
camera_sensor: Unknown symbol register_sensor
```

🧠 What really happens behind the scenes?

- **`EXPORT_SYMBOL(register_sensor)`** adds the symbol to the **kernel symbol table** (`/proc/kallsyms`)
- When **`camera_sensor.ko`** loads, the kernel:
  - Checks for missing symbols
  - Resolves **`register_sensor`** to the one in **`sensor_core.ko`**

❓ Can a module export data?

Yes — not just functions. You can also export variables:

```c
int sensor_count = 0;
EXPORT_SYMBOL(sensor_count);
```

Then in other module:

```c
extern int sensor_count;
sensor_count++;
```

⚠️ But be careful with locking — no protection by default.

🔁 Alternative: Bus/Subsystem

Some drivers don't export symbols directly. Instead, they **register** with a bus or subsystem.

For example:

```c
usb_register_driver(&my_usb_driver);
```

Other modules can then **discover** them or **bind** dynamically (like device probe logic).

✅ Summary

Feature	Description
`EXPORT_SYMBOL()`	Makes function or variable visible to others
`extern` in importer	Declares symbol from another module
`/proc/kallsyms`	Holds all exported symbols
`request_module()`	Dynamically loads a module by name (optional)
Subsystem registration	Used by USB, I2C, etc. for indirect communication

---

Would you like a ready-to-run full minimal **.ko module pair** to test on Android or Linux right now?

20. Yes

Awesome! Here's a **full working minimal example** of **two Linux kernel modules** that:

- Communicate via a function using **`EXPORT_SYMBOL()`**
- Can be compiled and tested on **Android (with root)** or any Linux system

---

📁 Files You Need

Create the following files in a folder:

1. `sensor_core.c` – (exports function)

```c
// sensor_core.c
#include <linux/module.h>
#include <linux/kernel.h>

void register_sensor(const char* name) {
    printk(KERN_INFO "sensor_core: Sensor registered: %s\n", name);
}
EXPORT_SYMBOL(register_sensor);  // 👈 make function available to others

static int __init sensor_core_init(void) {
    printk(KERN_INFO "sensor_core: Module loaded\n");
    return 0;
}

static void __exit sensor_core_exit(void) {
    printk(KERN_INFO "sensor_core: Module unloaded\n");
}

module_init(sensor_core_init);
module_exit(sensor_core_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("You");
MODULE_DESCRIPTION("Sensor Core Module");
```

---

2. `camera_sensor.c` – (calls function from other module)

```c
// camera_sensor.c
#include <linux/module.h>
#include <linux/kernel.h>

extern void register_sensor(const char* name);  // 👈 declare external symbol

static int __init camera_sensor_init(void) {
    printk(KERN_INFO "camera_sensor: Module loaded\n");
    register_sensor("Camera");
    return 0;
}

static void __exit camera_sensor_exit(void) {
    printk(KERN_INFO "camera_sensor: Module unloaded\n");
}

module_init(camera_sensor_init);
module_exit(camera_sensor_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("You");
MODULE_DESCRIPTION("Camera Sensor Module");
```

---

3. `Makefile`

```make
obj-m += sensor_core.o
obj-m += camera_sensor.o
```

```
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

🧪 How to Compile and Test

✅ 1. Compile

```bash
make
```

You'll get:

```
sensor_core.ko
camera_sensor.ko
```

✅ 2. Load and Test in Order

```bash
sudo insmod sensor_core.ko
sudo insmod camera_sensor.ko
```

Then check:

```bash
dmesg | tail -n 10
```

Expected output:

```csharp
[xxxx] sensor_core: Module loaded
[xxxx] camera_sensor: Module loaded
[xxxx] sensor_core: Sensor registered: Camera
```

⚠️ What if you reverse the order?

```bash
sudo rmmod sensor_core
sudo rmmod camera_sensor
sudo insmod camera_sensor.ko  # This will fail
```

Because:

```makefile
camera_sensor: Unknown symbol register_sensor
```

The kernel can't resolve the symbol unless **`sensor_core`** is loaded first.

✅ Summary

Step	Description
`EXPORT_SYMBOL()`	Makes a function available to others

Step	Description
`extern`	Used in the calling module
`insmod` order	Core module must be loaded first
`dmesg`	Shows debug logs from both modules

Would you like me to help adapt this to Android kernel module (NDK kernel source, **`.ko`** signing, etc.) or integrate with a device HAL?

21. Syntax of dts

Sure! Let's explain the **syntax of a `.dts` (Device Tree Source)** file in a very simple and clear way, step-by-step.

📘 What is a `.dts` file?

A **`.dts`** file describes the **hardware layout** of a board or device in a tree-like structure.
It's written in a special C-like syntax, and compiled into a **`.dtb`** binary file.

📐 Basic Syntax Structure

Here is the **template structure** of a **`.dts`** file:

```dts
/dts-v1/;                          // required header for device tree version 1

/include/ "filename.dtsi"         // include a shared .dtsi file (optional)

 / {                               // Root node
    compatible = "vendor,device"; // Required string identifying the board

    model = "MyBoard v1.0";        // Optional human-readable name

    aliases {
        serial0 = &uart0;
    };

    chosen {
        bootargs = "console=ttyS0,115200";
    };

    memory@80000000 {
        device_type = "memory";
        reg = <0x80000000 0x40000000>;  // Start address and size (1 GB)
    };

    uart0: serial@10000000 {
        compatible = "ns16550a";
        reg = <0x10000000 0x1000>;
        clock-frequency = <24000000>;
        status = "okay";
    };

    gpio_keys {
        compatible = "gpio-keys";
        button@0 {
            label = "power";
            gpios = <&gpio1 3 0>;   // GPIO controller, pin, flags
            linux,code = <116>;    // KEY_POWER
        };
    };
};
```

🔍 Explanation of Each Element

Syntax	Meaning
`/dts-v1/;`	Declares this file uses Device Tree v1 format
`/include/ "file.dtsi"`	Includes common/shared definitions (like C **`#include`**)
`/ { ... };`	Root node — defines global and top-level devices
`compatible = "vendor,device"`	Unique string used to match driver with this hardware node
`reg = <addr size>`	Describes physical memory-mapped address and size
`status = "okay"`	Marks the device as enabled (**`"disabled"`** means it won't be used)
`node-name@address {}`	Declares a device or sub-node at specific address
`label:`	Named label so other nodes can refer using **`&label`**
`&label`	Refers to a previously declared node
`gpios = <...>`	GPIO controller phandle, pin number, and flags (direction, active low)

Syntax	Meaning
`bootargs`	Kernel boot parameters (command line)
`linux,code`	Key code from Linux input subsystem

🧱 Node Definition Syntax

```dts
node-name@unit-address {
    property-name = value;
    property2 = <hex_value>;
    subnode@subaddr {
        ...
    };
};
```

Example:

```dts
ethernet@1c50000 {
    compatible = "realtek,rtl8211";
    reg = <0x1c50000 0x1000>;
    interrupts = <23>;
    phy-handle = <&phy0>;

    phy0: ethernet-phy@0 {
        reg = <0>;
    };
};
```

📦 Value Types in DTS

Syntax Example	Meaning
`"text"`	A string
`<0x1234 0x5678>`	Hexadecimal numbers
`<1>` or `<0>`	Integer (decimal)
`[12 34 ab cd]`	Byte array (for MAC address, etc.)
`&label`	Reference to another node

✅ Summary

Element	Description
`node@addr`	Declares a device or region
`reg`	Defines memory address + size for device
`compatible`	Tells kernel which driver to use
`status`	`"okay"` to enable, `"disabled"` to skip
`label:`	Gives a name so other nodes can reference it
`&label`	Used to refer to another node

Would you like a **cheat sheet or a diagram** that maps this syntax visually? Or maybe want to see a **real `.dts` used on an Android phone**?

22. ↪
`reg = <addr size>`
what is this

Great — you're asking about this line in a Device Tree:

```dts
reg = <addr size>;
```

Let's explain it very simply and clearly:

✅ What is `reg` in Device Tree?

The **`reg`** property tells the **physical memory address** and **size** of the hardware device.
It's how the kernel knows:

> "This device lives at memory address **X** and takes up **Y** bytes."

🔍 Syntax

```dts
reg = <address size>;
```

Element	Meaning
`address`	Where in memory this device is mapped (start address)
`size`	How much memory the device uses

✅ Example:

```dts
reg = <0x10000000 0x1000>;
```

- **`0x10000000`** → Start address in physical memory
- **`0x1000`** (4096 bytes) → Size of the device's memory range (4 KB)

💬 This means:

> "This hardware device is located at address **`0x10000000`** and uses 4 KB of memory."

🧠 Where does this info come from?

This info is given by:

- SoC vendor (Qualcomm, MediaTek, etc.)
- Hardware manual or reference manual
- Platform documentation

It matches how the device is memory-mapped by the SoC.

📚 Multi-Address Example

Sometimes **`reg`** can define **multiple address/size pairs**:

```dts
reg = <0x10000000 0x1000>, <0x20000000 0x1000>;
```

This means:

- First memory region: **`0x10000000`** (size 4 KB)
- Second memory region: **`0x20000000`** (size 4 KB)

This is used if a device needs access to multiple memory blocks.

📏 Word Size and Cells

The `<...>` brackets hold 32-bit or 64-bit values (depends on **`#address-cells`** and **`#size-cells`** from the parent bus). But normally:

```dts
reg = <address size>;  // Both 32-bit
```

Some platforms (like 64-bit ARM) may use:

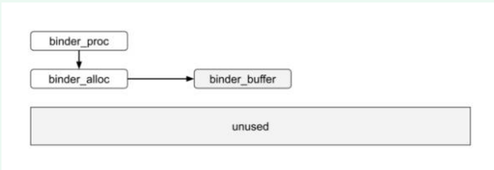
```dts
reg = <0x0 0x10000000 0x0 0x1000>;  // 64-bit addr + size (split in 2 each)
```

✅ Summary

Field	Meaning
`reg`	Describes where device lives in memory
`address`	Start of memory-mapped I/O region
`size`	Length of memory used by the device
Format	**`<addr size>`** (usually 32-bit)

Would you like to see how to find a real device's **`reg`** values from Linux or how to match it with SoC datasheet/registers?

23.



This diagram is showing part of the **Android Binder kernel driver's buffer allocation system** — specifically how Binder allocates memory for inter-process communication (IPC).

Let me break it down in **simple words**:

🔍 Components in the Diagram

📦 `binder_proc`

- Represents a **process** in the Binder system (e.g., an app or system service).
- Holds per-process state (like buffer space, thread list, etc.).

🧠 `binder_alloc`

- This is the **allocator** used by that process (`binder_proc`) to manage a chunk of memory.
- Responsible for allocating and freeing memory for IPC transactions.
- Think of this as a memory manager for one Binder process.

📄 `binder_buffer`

- Represents one **allocated buffer** inside the shared memory area.
- This buffer is used to hold transaction data (like parcels or objects) that one process is sending to another.

▯ `unused` region (gray box)

- This is the **remaining free space** inside the preallocated shared memory area.
- When a new transaction happens, the allocator tries to find free space (like here) to place a new `binder_buffer`.

🧱 What this diagram tells us

1. A `binder_proc` has a `binder_alloc` region (memory manager).
2. `binder_alloc` holds `binder_buffer` objects — each representing allocated memory for IPC messages.
3. After the used buffers, there's still **unused memory** — available for future allocations.

This relates to **Binder's best-fit allocation strategy**, where it tries to fit new buffers into the smallest available gaps in memory.

📦 Real-life scenario

Let's say:

- App A sends a large object to System Server
- Binder allocates a `binder_buffer` from shared memory (inside `binder_alloc`)
- If more data comes in, it checks if more space (unused) is available

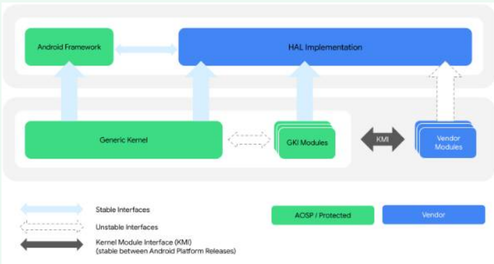
If memory runs out or becomes fragmented, the Binder driver might:

- Return an error (ENOMEM)
- Or force the app to wait until some buffers are freed

Would you like to go deeper into:

- How Binder manages memory using red-black trees?
- Or what triggers **`binder_alloc_new_buf()`** internally?

24.



The Android kernel is based on an upstream Linux Long Term Supported (LTS) kernel. At Google, LTS kernels are combined with Android-specific patches to form what are known as Android Common Kernels (ACKs).

Newer ACKs (version 5.4 and above) are also known as GKI kernels. GKI kernels support the separation of the hardware-agnostic generic core kernel code and GKI modules from the hardware-specific vendor modules.

The GKI kernel interacts with hardware-specific vendor modules containing system on a chip (SoC) and board-specific code. The interaction between the GKI kernel and vendor modules is enabled by the Kernel Module Interface (KMI) consisting of symbol lists identifying the functions and global data required by vendor modules. Figure 1 shows the GKI kernel and vendor module architecture:

Great! You're looking at the **GKI (Generic Kernel Image) architecture diagram**, which is part of how modern Android handles kernel modularity and vendor separation. Let's explain this image **in simple terms**, block by block:

📷 What You're Seeing in the Diagram

This diagram explains how Android separates its **generic kernel** and **vendor-specific code**, which helps make updates easier and modular.

🧱 Main Layers in the Diagram

1. Android Framework (Top Layer)

- This is Android Java/Kotlin framework APIs (what apps and system services use)
- Communicates with HALs (Hardware Abstraction Layers) using Binder IPC

---

2. HAL Implementation

- This is the **vendor-provided C/C++ code** that sits between the framework and the hardware.
- It talks to the kernel drivers or vendor modules via syscalls or ioctls
- Examples: `camera.default.so`, `sensors.default.so`, etc.

---

3. Generic Kernel (Green)

- The mainline kernel part — common to all devices.
- Provided by Google, built from LTS (Long-Term Support) Linux kernels + Android patches
- Also called **ACK** or **GKI** kernel (for version 5.4+)

---

4. GKI Modules (Green Boxes)

- Loadable modules that are generic and provided by AOSP
- Example: `binder.ko`, `ashmem.ko`, etc.
- These can be dynamically loaded into the kernel as needed

---

5. Vendor Modules (Blue Boxes)

- These are **hardware-specific modules** from OEMs or SoC vendors (Qualcomm, MediaTek, etc.)
- Examples: GPU drivers, camera drivers, modem drivers
- These are not part of GKI, but they **plug into it** using a stable interface

6. KMI (Kernel Module Interface)

- This is the **contract or API boundary** between the GKI and vendor modules.
- Google maintains a **symbol list** (list of allowed kernel functions) so vendor modules know what's safe to call.

🔁 Arrows & Colors

Arrow	Meaning
🔵 Solid Blue	Interface between layers (e.g. Framework ↔ HAL, HAL ↔ Kernel)
⚫ Dashed Black	Interaction with vendor components
🟢 Green	Provided by Google (AOSP)
🔵 Blue	Provided by Vendor/OEM (SoC-specific)

✅ Why This Architecture Exists?

1. **Modular Updates**: GKI is the same for all devices → Easier OTA kernel updates
2. **Vendor Separation**: Vendor drivers are in `.ko` files and built separately
3. **Stability**: Vendor code only calls allowed functions via KMI
4. **Security**: Tighter control over what vendor modules can do

🧠 Summary Table

Layer	Who Provides It	Purpose
Android Framework	Google (AOSP)	API for system services and apps
HAL Implementation	Vendor	Talks to hardware drivers
GKI Kernel	Google	Generic Android kernel based on Linux LTS
GKI Modules	Google	Common loadable kernel modules
Vendor Modules	Vendor (OEM/SoC)	SoC/Board-specific kernel modules (`.ko`)
KMI	Google (defined)	Interface for vendor drivers to use safely

Would you like to explore:

- Where these `.ko` files are stored in Android?
- How to build or debug a vendor module?
- Or how KMI symbol lists are maintained?

25. ↳
KMI (Kernel Module Interface)
explain simple words

26. ↪
KMI (Kernel Module Interface)
why we use this and the benefits

Let's explain **KMI (Kernel Module Interface)** in **very simple words**, and **why we use it** in Android:

🧠 What is KMI?

KMI is like a **fixed agreement or contract** between:

- **The Android Generic Kernel (GKI)** – made by Google
- **Vendor drivers/modules** – made by Qualcomm, MediaTek, etc.

It tells vendors:

> "Here is the exact list of functions and global variables your drivers are allowed to use. Don't touch anything else."

📦 Why is this needed?

Because **Google wants to update the Android kernel without breaking vendor drivers.**

If the kernel changes randomly, your vendor drivers may stop working.
So KMI says:

> "We promise to keep this interface stable — just use these functions."

✅ Benefits of KMI

Benefit	Simple Explanation
✅ **Modular updates**	GKI can be updated without breaking vendor drivers
✅ **Separation of concerns**	Google builds the kernel, vendors build drivers independently
✅ **Security**	Vendors can't touch dangerous or internal kernel stuff
✅ **Stability**	Vendor drivers won't break when kernel is updated
✅ **Reusability**	Same kernel works on many devices with different vendors

---

🎯 Example Analogy

Imagine:

- **Google is the government**, building a public highway (the kernel)
- **Vendors are the truck drivers**, delivering goods (drivers)
- **KMI is the rulebook**: "Drive only on this lane, at this speed, use this bridge"

This way:

- Google can fix or repave roads (update kernel) without breaking traffic (vendor drivers)

🧩 Without KMI? (Before GKI)

- Every vendor customized the kernel
- Hard to upgrade
- Required full re-testing
- Security holes appeared

- Android fragmentation

✅ Summary in 1 Sentence:

KMI is a fixed list of allowed kernel functions that vendor drivers can use, so that Google can safely update the Android kernel without breaking vendor code.

Would you like to see:

- Where the KMI list is defined in the source?
- Or how to make your own `.ko` compatible with KMI?

27. GKI modules vs Vendor modules

Great question! Let's clearly and simply compare **GKI Modules** vs **Vendor Modules** in Android 👇

🧱 First, what are modules?

Kernel modules (`.ko` files) are **pieces of the kernel** that can be **loaded/unloaded at runtime**.
They usually provide drivers (e.g. for camera, GPU, modem, etc.)

⚔️ GKI Modules vs Vendor Modules

Feature	GKI Modules	Vendor Modules
🔧 Who builds them?	**Google / AOSP**	**Device/SOC Vendor** (e.g. Qualcomm, MTK)
🧠 What are they for?	**Generic features** shared by all devices	**Device-specific drivers** (hardware-bound)
📦 Stored in which partition?	`/system/lib/modules`	`/vendor/lib/modules`
🔗 Use KMI?	Yes	Yes (must follow KMI to work with GKI)
🔄 Can be updated via OTA?	Yes (as part of GKI update)	Yes (as part of vendor update)
🔍 Examples	`binder.ko`, `dm-verity.ko`, `ashmem.ko`	`msm_camera.ko`, `qcom_wlan.ko`, `gpu.ko`

✅ GKI Modules (Generic Kernel Image)

- Built by Google
- Common across all Android devices
- You can think of these as "standard Linux parts with Android support"

Example:

- `binder.ko` – binder IPC driver
- `dm-verity.ko` – verified boot
- `ion.ko` – memory allocator

✅ Vendor Modules

- Made by the SoC or device manufacturer (Qualcomm, Samsung, MediaTek, etc.)
- Contain board or chip-specific code
- Must follow the **KMI rules** (can't access anything outside allowed symbols)

Example:

- `msm_camera.ko` – for Qualcomm camera hardware
- `gpu_mali.ko` – Mali GPU driver
- `qcom_wlan.ko` – Wi-Fi driver for Qualcomm chip

🧠 Why this separation matters?

Why?	Benefit
🔁 Easier updates	Google can update GKI without touching vendor drivers
🔐 Security	GKI is controlled, stable, and verified
🧩 Compatibility	Same kernel works across many devices, only vendor modules change
🔧 Modularity	Device vendors can build and update their drivers independently

🎯 Analogy:

Role	GKI Modules	Vendor Modules
Like a	Android OS core components	Hardware plug-in drivers
Made by	Google	Qualcomm / MediaTek / OEM
Target	All phones	One specific device or SoC

✅ Summary

> **GKI Modules** are generic kernel drivers built by Google for all Android devices.
> **Vendor Modules** are device-specific drivers built by the vendor (like Qualcomm), which plug into the GKI using the KMI.

Would you like to list all the GKI modules from a real device or learn how to build a custom vendor module?

28. Kernel glossary
This section contains terms used throughout the kernel documentation.

Kernel types
Android Common Kernel (ACK)
A kernel that is downstream of a LTS kernel and includes patches of interest to the Android community that haven't been merged into Linux mainline or Long Term Supported (LTS) kernels. Newer ACKs (version 5.4 and above) are also known as GKI kernels as they support the separation of hardware agnostic Generic Kernel code and hardware agnostic GKI modules.
Android Open Source Project (AOSP) kernel
See Android Common Kernel.
Feature Kernel
A kernel for which the platform release features are guaranteed to be implemented. For example, in Android 12, the two feature kernels were android12-5.4 and android12-5.10. Android 12 features can't be backported to 4.19 kernels; the feature set would be similar to a device that launched with 4.19 on Android 11 and upgraded to Android 12.
generic core kernel
The portion of the GKI kernel that is common across all devices.
Generic Kernel Image (GKI) kernel
Any newer (5.4 and above) ACK kernel (currently aarch64 only). This kernel has two parts: The core GKI kernel with code common across all devices and GKI modules developed by Google that can be dynamically loaded on devices where applicable.
Kernel Module Interface (KMI) kernel
See GKI kernel.
Launch Kernel
A kernel that is valid for launching a device for a given Android platform release. For example, in Android 12, the valid launch kernels were 4.19, 5.4, and 5.10.
Long Term Supported (LTS) kernel
A Linux kernel that's supported for 2 to 6 years. LTS kernels are released once per year and are the basis for each of Google's ACKs.
Branch types
ACK KMI kernel branch
The branch for which GKI kernels are built. For example, android12-5.10 and android13-5.15.
Android-mainline
The primary development branch for Android features. When a new LTS kernel is declared upstream, the corresponding new GKI kernel is branched from android-mainline.
Linux mainline
The primary development branch for the upstream Linux kernels, including LTS kernels.
Other terms
Certified boot image
The kernel delivered in binary form (boot.img) and flashed onto the device. This image is certified in the sense that it contains embedded certificates so Google can verify that the device ships with a kernel certified by Google.
Dynamically loadable kernel module (DLKM)
A module that can be dynamically loaded during device boot depending on the needs of the device. GKI and vendor modules are both types of DLKMs. DLKMs are released in .ko form and can be drivers or can deliver other kernel functionality.
GKI project
A Google project addressing kernel fragmentation by separating common core kernel functionality from vendor-specific SoC and board support into loadable modules.
Generic Kernel Image (GKI)
A boot image certified by Google that contains a GKI kernel built from an ACK source tree and is suitable to be flashed to the boot partition of an Android device.
KMI
An interface between the GKI kernel and vendor modules allowing vendor modules to be updated independently of the GKI kernel. This interface consists of kernel functions and global data that have been identified as vendor/OEM dependencies using per-partner symbol lists.
GKI modules
Kernel modules built by Google that can be dynamically loaded on devices where applicable. These modules are built as artifacts of the GKI kernel and are delivered alongside GKI as the system_dlkm_staging_archive.tar.gz archive. GKI modules are signed by Google using the kernel build time key pair and are compatible only with the GKI kernel that they're built with.
Vendor module
A hardware-specific module developed by a partner and that contains SoC and device-specific functionality. A vendor module is a type of dynamically loadable kernel module.

Here's a **simplified and easy-to-understand summary** of the Android **Kernel Glossary** you posted — with all key terms explained like you're learning from scratch.

🔧 Kernel Types

Term	Meaning
Android Common Kernel (ACK)	A modified version of the Linux Long-Term Support (LTS) kernel, maintained by Google. It includes patches specific to Android that haven't yet made it to Linux mainline.
GKI Kernel (Generic Kernel Image)	A newer (Android 10+, kernel 5.4+) version of ACK that separates **common core** and **modular parts**. Built to work on **any Android device**.
Generic Core Kernel	The main part of the GKI kernel – same for all devices.
Feature Kernel	A kernel version guaranteed to support all Android features for a given release (e.g., **`android12-5.10`**).
Launch Kernel	A kernel version allowed to be used to **launch** new devices in a given Android release (e.g., 4.19, 5.4, 5.10 for Android 12).
LTS Kernel (Long Term Supported)	A kernel from upstream Linux with 2–6 years of support. Google starts with this and builds ACK.
AOSP Kernel	Same as ACK — kernel shared via AOSP source tree.

🌿 Branch Types

Term	Meaning
ACK KMI Kernel Branch	Where GKI kernels are maintained. Examples: **`android13-5.15`**, **`android12-5.10`**.
android-mainline	Google's working branch where Android kernel features are developed before moving to ACK.
Linux mainline	The official Linux kernel development branch maintained by Linus Torvalds and the community. Google pulls updates from here.

📦 Modules and Interfaces

Term	Meaning
GKI Modules	Kernel **`.ko`** modules built by Google, for features like **`binder`**, **`ion`**, etc. They're part of the GKI but loaded separately (DLKM).
Vendor Modules	Hardware-specific **`.ko`** modules built by chip vendors (e.g., Qualcomm, MediaTek) for things like GPU, camera, modem.
DLKM (Dynamically Loadable Kernel Module)	A **`.ko`** file that gets loaded during boot depending on device need. GKI and vendor modules are both DLKMs.
KMI (Kernel Module Interface)	A stable list of kernel symbols (functions, variables) that vendor modules are **allowed to use**. Google promises not to break this list.
Certified Boot Image	The final **`boot.img`** (with GKI kernel) that is signed and certified by Google — ensuring the kernel is safe and verified.

📁 What is the GKI Project?

Google's **GKI project** is an effort to **stop kernel fragmentation** by:

- Keeping the **core kernel common** across devices
- Letting vendors plug in hardware support through **modules**
- Allowing **faster updates** and better **security**

🎯 Why All This Matters?

Problem (Before)	Solution (Now with GKI)
Every vendor used a custom kernel	One shared core kernel (GKI)
Hard to update kernels	Easy to update core, keep vendor drivers
Insecure drivers could access anything	KMI restricts what vendor modules can use
Vendor lock-in to specific Android versions	Now kernel can be updated independently

✅ Summary

Term	Simple Explanation
ACK	Android kernel based on LTS + Android patches

Term	Simple Explanation
GKI	Common Android kernel with modularity support
GKI Modules	Google-built **`.ko`** files, loadable by GKI
Vendor Modules	SoC/vendor-specific drivers (**`.ko`**)
DLKM	Any **`.ko`** file that loads dynamically
KMI	A stable interface for vendor drivers
Certified Boot	Google-signed boot image with GKI kernel

Would you like a **diagram** showing all this visually (boot.img → GKI → modules → vendor)? Or want help locating or building a GKI module on your system?

29. can you types of kernel u mentioned and why they exist?

Absolutely! Let's go through the **types of kernels in Android** that were mentioned in the glossary and **explain why each type exists** in **very simple words**.

🔧 1. LTS Kernel (Long Term Supported)

🟢 **What it is**:

- A **stable Linux kernel** maintained by the Linux community (e.g., `5.4`, `5.10`, `5.15`).
- Supported for **2–6 years** with bug fixes and security updates.

🧠 **Why it exists**:

- To give a reliable foundation for operating systems like Android.
- Android doesn't use cutting-edge kernels — it starts from these safe, tested LTS versions.

🔧 2. Android Common Kernel (ACK)

🟢 **What it is**:

- A Linux LTS kernel + **Android-specific patches** (like Binder, Ashmem, etc.).
- Maintained by Google.

🧠 **Why it exists**:

- To add **Android features** that don't exist in upstream Linux.
- All Android devices start from this base instead of raw LTS.

🔧 3. GKI Kernel (Generic Kernel Image)

🟢 **What it is**:

- A **special type of ACK (version 5.4+)**.
- It splits the kernel into:
  - A **generic core** (same for all devices)
  - And **modules** (loadable `.ko` files)

🧠 **Why it exists**:

- To make Android kernels **modular**, **updatable**, and **less fragmented**.
- Vendors just plug in their own modules — no need to modify the core kernel.

🔧 4. Feature Kernel

🟢 **What it is**:

- A kernel version that supports **all Android features** for a particular Android version.

📘 **Example**:

- Android 12 feature kernels: `android12-5.4`, `android12-5.10`

🧠 **Why it exists**:

- To ensure all devices running Android 12 get the expected features, security, and behavior.

🔧 5. Launch Kernel

🟢 **What it is**:

- A kernel version **approved by Google** to launch **new devices** for a specific Android release.

📘 **Example**:

- For Android 12: launch kernels = `4.19`, `5.4`, `5.10`

🧠 **Why it exists**:

- To keep Android versions **consistent and testable** across OEMs.
- Google can ensure new devices meet the latest standards.

🔧 6. AOSP Kernel

🟢 **What it is**:

- The kernel source **published on AOSP** (source.android.com).
- Usually refers to the **ACK**, but visible to the public.

🧠 **Why it exists**:

- For developers and vendors to access Android-specific kernel code.
- Helps in building custom ROMs, kernel modules, etc.

🔧 7. KMI Kernel

🟢 **What it is**:

- Another name for GKI kernel — emphasizes the use of a **Kernel Module Interface (KMI)**.

🧠 **Why it exists**:

- To allow vendors to write `.ko` drivers **without breaking** when the kernel updates.
- The KMI ensures **backward compatibility** between core kernel and vendor modules.

📊 Summary Table

Kernel Type	Based On	Purpose
LTS Kernel	Upstream Linux	Reliable foundation with long support
ACK	LTS	Android-specific kernel with Binder, Ashmem, etc.
GKI	ACK (5.4+)	Modular kernel split into generic + loadable parts
Feature Kernel	GKI	Guaranteed to support Android features for a release
Launch Kernel	GKI/ACK	Approved kernel versions for launching new devices
AOSP Kernel	ACK	Public version of ACK in Android source tree
KMI Kernel	GKI	Same as GKI, focused on vendor-module compatibility (symbol interface)

✅ Why So Many?

Because Google needs to:

- Start with something stable (LTS)
- Add Android features (ACK)
- Ensure device compatibility (GKI)

- Let vendors keep drivers private (via KMI)
- Keep everything **secure**, **modular**, and **updatable**

Would you like a **visual diagram** showing how LTS → ACK → GKI → Vendor Modules flows?
Great! Let's clear up the **confusion between Feature Kernel and Launch Kernel** with simple explanations and examples. These are Android-specific terms used by Google to control kernel support across versions.

🎯 The Key Difference

Term	Simple Meaning	Used For
Feature Kernel	A kernel that supports **all new Android platform features** (e.g., new APIs, security features, etc.) for a specific Android version	For **devices upgrading** or **testing new features**
Launch Kernel	A kernel that is **allowed to launch new devices** with a specific Android release	For **new phones** launching with that Android version

---

📘 Example: Android 12

Let's say Android 12 supports:

- New power management
- New memory features
- And security enhancements

Google chooses these kernel versions:

Kernel	Role
`4.19`	Launch kernel only (limited features)
`5.4`	Launch + Feature kernel (full Android 12 support)
`5.10`	Launch + Feature kernel (full support)

✅ Launch Kernel

- Minimum requirement to **ship a new phone** on Android 12.
- Might not support **all Android 12 features**.
- Devices **can** use older kernels like `4.19` to **reduce effort** (e.g., reuse old SoC BSPs).

⏩ *Used by OEMs to quickly launch new models.*

✅ Feature Kernel

- Fully supports **ALL new Android features** of that release.
- Used in:
  - Devices that **want full feature set**
  - **Upgrades** from older Android versions
  - **Development/testing**

⏩ *Used to ensure feature parity, test GKI changes, etc.*

🔍 Summary Table

Feature	Launch Kernel	Feature Kernel
Purpose	Allowed to launch new devices	Guarantees support for new features
Required for new phones?	✅ Yes	❌ Optional (but recommended)
Android features	May not support all	Must support all

Feature	Launch Kernel	Feature Kernel
Example Use	Shipping phones quickly	Testing, upgrades, feature completeness
Kernel version	Can be older (`4.19`, `5.4`)	Must be newer (`5.4`, `5.10`, etc.)

📦 Analogy

> Imagine you're making Android cars 🚗

- **Launch Kernel** = The **engine types you're allowed to use** to release a new model.
- **Feature Kernel** = The **engine types that support all latest features**, like autopilot or eco-mode.

You **can launch a car with an older engine**, but you **won't get all new features**.

Let me know if you'd like a diagram showing how these kernel types relate across Android versions (11, 12, 13, etc.)

30. still confused

Got it — let's explain it again even **more simply**, using real-world words and analogies 👇

🔧 Imagine: Android as a Car Company

You're Google. You design **Android** like a **car**. The **engine** is the **Linux kernel**.

Every year, you release a new **Android version** (Android 12, 13, 14...) like a **new car model**.

But you don't build the whole car. You let others (Samsung, Xiaomi, etc.) build their own cars (phones) using your Android design.

Now comes the **kernel choice**.

🚀 1. What is a Launch Kernel?

🟢 **Think: Minimum engine required to sell a new car this year**

- Google says: "If you want to launch a **new phone** with Android 12, you **must** use kernel version X, Y, or Z."
- Example for Android 12: allowed launch kernels are `4.19`, `5.4`, `5.10`.

💬 Why?
So Google can test and approve just a few kernels. Easier for security and updates.

✅ Used when:

- Making a **brand new phone**
- You **want to ship fast**, using older drivers or SoC BSP

🧠 2. What is a Feature Kernel?

🟢 **Think: Engine that supports all the newest car features**

- Google says: "If you want all Android 12 features to work, your kernel must support them."
- These are the **Feature Kernels**.

💬 Why?
Some Android features need support from the kernel (like memory tagging, new scheduling, etc.)

✅ Used when:

- You're **upgrading an old phone** to Android 12
- Or want **full Android features**

🧠 Clear Analogy

Term	Real-world analogy	Purpose
Launch Kernel	Minimum engine allowed to sell a car	For launching **new phones**
Feature Kernel	Engine that supports all new features	For enabling **all Android features**

✅ Summary Table

Feature	Launch Kernel	Feature Kernel
Who uses it?	Phone makers (OEMs)	OEMs + Google for testing
Purpose	Launch new phones	Test or upgrade with full features
Android version tied?	Yes	Yes

Feature	Launch Kernel	Feature Kernel
Includes all features?	❌ Not guaranteed	✅ Yes
Required for launch?	✅ Yes	❌ Optional

📘 Example from Android 12

Kernel Version	Launch Kernel?	Feature Kernel?
`4.19`	✅ Yes	❌ No (missing features)
`5.4`	✅ Yes	✅ Yes
`5.10`	✅ Yes	✅ Yes

👇 Final Takeaway

> **To launch a new phone**, you need a **Launch Kernel** (approved list).
> **To support all Android features**, you should use a **Feature Kernel**.

You can **launch with a Launch Kernel** even if it's **not a Feature Kernel**, but you'll **miss out on some new features**.

Would a **visual diagram** or **timeline chart** help? Or want to see what kernel your device uses and whether it's a feature or launch kernel?

31. ↪
Feature Kernel
explain with example

Perfect — let's walk through a **real-world example** to make the concept of a **Feature Kernel** crystal clear.

📱 Example: Android 12

Google released Android 12 in 2021.

They said:

> "The **feature kernels** for Android 12 are:
>
> - `android12-5.4`
> - `android12-5.10`"

These are the **only kernels** that support **all Android 12 features**, such as:

- Memory Tagging Extension (MTE)
- New power management policies
- Improved scheduler changes
- New security features like `fs-verity`

📲 Two Example Devices

✅ Device A (e.g., Pixel 5)

- Uses **kernel 5.4**
- Built with `android12-5.4` branch
- It is a **feature kernel**
- So, this device can use **100% of Android 12's capabilities**

➡️ Example: full support for **MTE**, **fs-verity**, etc.

⚠️ Device B (e.g., older Samsung phone)

- Was launched on Android 11 with **kernel 4.19**
- Samsung wants to **upgrade** it to Android 12 (via OTA)
- But kernel 4.19 is **not** a feature kernel for Android 12

So what happens?

- The upgrade works ✅
- But: Some Android 12 features will **not be available** ❌

➡️ Example: MTE might be missing, some new power management features won't work.

✅ What Feature Kernel means (in this case)

Device	Kernel	Android 12 Features	Feature Kernel?
Pixel 5	5.4	✅ All features	✅ Yes
Samsung A51	4.19	❌ Some missing	❌ No

📦 Summary

> A **Feature Kernel** is a kernel version that is guaranteed by Google to support **all features** of a specific Android release.

So when:

- **Launching** a new phone → use **Launch Kernel**
- **Upgrading** an old phone → you **can** use older kernel, but you **won't get all features** unless it's a Feature Kernel

---

Would you like to check if your phone uses a **feature kernel**, or want the full list of Android versions and their feature kernel versions?

32. Boot time optimization

bookmark_border
This page provides tips to improve boot time.

Strip debug symbols from modules
Similar to how debug symbols are stripped from the kernel on a production device, make sure you also strip the debug symbols from modules. Stripping debug symbols from modules helps boot time by reducing the following:

The time it takes to read the binaries from flash.
The time it takes to decompress the ramdisk.
The time it takes to load the modules.
Stripping debug symbol from modules may save several seconds during boot.

Symbol stripping is enabled by default in the Android platform build, but to explicitly enable them, set BOARD_DO_NOT_STRIP_VENDOR_RAMDISK_MODULES in your device-specific config under device/vendor/device.

Use LZ4 compression for kernel and ramdisk
Gzip generates a smaller compressed output compared to LZ4, but LZ4 decompresses faster than Gzip. For the kernel and modules, the absolute storage size reduction from using Gzip isn't that significant compared to the decompression time benefit of LZ4.

Support for LZ4 ramdisk compression has been added to the Android platform build through BOARD_RAMDISK_USE_LZ4. You can set this option in your device-specific config. Kernel compression can be set through kernel defconfig.

Switching to LZ4 should give 500ms to 1000ms faster boot time.

Avoid excessive logging in your drivers
In ARM64 and ARM32, function calls that are more than a specific distance from the call site need a jump table (called a procedure linking table, or PLT) to be able to encode the full jump address. Since modules are loaded dynamically, these jump tables need to be fixed up during module load. The calls that need relocation are called relocation entries with explicit addends (or RELA, for short) entries in the ELF format.

The Linux kernel does some memory size optimization (such as cache hit optimization) when allocating the PLT. With this upstream commit, the optimization scheme has an O(N^2) complexity, where N is the number of RELAs of type R_AARCH64_JUMP26 or R_AARCH64_CALL26. So having fewer RELAs of these types is helpful in reducing the module load time.

One common coding pattern that increases the number of R_AARCH64_CALL26 or R_AARCH64_JUMP26 RELAs is excessive logging in a driver. Each call to printk() or any other logging scheme typically adds a CALL26/JUMP26 RELA entry. In the commit text in the upstream commit, ,notice that even with the optimization, the six modules take about 250ms to load—that is because those six modules were the top six modules with the most amount of logging.

Reducing logging can save can save about 100 - 300ms on boot times depending on how excessive the existing logging is.

Enable asynchronous probing, selectively
When a module is loaded, if the device that it supports has already been populated from the DT (devicetree) and added to driver core, then the device probe is done in the context of the module_init() call. When a device probe is done in the context of module_init(), the module can't finish loading until the probe completes. Since module loading is mostly serialized, a device that takes a relatively long time to probe slows the boot time.

To avoid slower boot times, enable asynchronous probing for modules that take a while to probe their devices. Enabling asynchronous probing for all modules might not be beneficial as the time it takes to fork a thread and kick off the probe might be as high as the time it takes to probe the device.

Devices that are connected through a slow bus such as I2C, devices that do firmware loading in their probe function, and devices that do a lot of hardware initialization can lead to the timing issue. The best way to identify when this happens is to collect the probe time for every driver and sort it.

To enable asynchronous probing for a module, it isn't sufficient to only set the PROBE_PREFER_ASYNCHRONOUS flag in the driver code. For modules, you also need to add module_name.async_probe=1 in the kernel command line or pass async_probe=1 as a module parameter when loading the module using modprobe or insmod.

Enabling asynchronous probing can save about 100 - 500ms on boot times depending on your hardware/drivers.

Probe your CPUfreq driver as early as possible

The earlier your CPUfreq driver probes, the sooner you can scale the CPU frequency to maximum (or some thermally limited maximum) during boot. The faster the CPU, the faster the boot. This guideline also applies to devfreq drivers that control the DRAM, memory, and interconnect frequency.

With modules, the load ordering can depend on the initcall level and compile or link order of the drivers. Use an alias MODULE_SOFTDEP() to make sure the cpufreq driver is among the first few modules to load.

Apart from loading the module early, you also need to make sure all the dependencies to probe the CPUfreq driver have also probed. For example, if you need a clock or regulator handle to control the frequency of your CPU, make sure they are probed first. Or you might need thermal drivers to be loaded before the CPUfreq driver if it is possible for your CPUs to get too hot during boot up. So, do what you can to make sure the CPUfreq and relevant devfreq drivers probe as early as possible.

The savings from probing your CPUfreq driver early can be very small to very large depending on how early you can get these to probe and at what frequency the bootloader leaves the CPUs in.

Move modules to second stage init, vendor or vendor_dlkm partition
Because the first stage init process is serialized, there aren't many opportunities to parallelize the boot process. If a module isn't needed for first stage init to finish, move the module to second stage init by placing it in the vendor or vendor_dlkm partition.

First stage init doesn't require probing several devices to get to second stage init. Only console and flash storage capabilities are needed for a normal boot flow.

Load the following essential drivers:

watchdog
reset
cpufreq
For recovery and user space fastbootd mode, first stage init requires more devices to probe (such as USB), and display. Keep a copy of these modules in the first stage ramdisk and in the vendor or vendor_dlkm partition. This lets them be loaded in first stage init for recovery or fastbootd boot flow. However, don't load the recovery mode modules in first stage init during normal boot flow. Recovery mode modules can be deferred to second stage init to decrease the boot time. All other modules that aren't needed in first stage init should be moved to the vendor or vendor_dlkm partition.

Given a list of leaf devices (for example, the UFS or serial), dev needs.sh script finds all drivers, devices, and modules needed for dependencies or suppliers (for example, clocks, regulators, or gpio) to probe.

Moving modules to second stage init decreases boot times in the following ways:

Ramdisk size reduction.
This yields faster flash reads when the bootloader loads the ramdisk (serialized boot step).
This yields faster decompression speeds when the kernel decompresses the ramdisk (serialized boot step).
Second stage init works in parallel, which hides the module's loading time with the work being done in second stage init.
Moving modules to second stage can save 500 - 1000ms on boot times depending on how many modules you're able to move to second stage init.

Module loading logistics
The latest Android build features board configurations that control which modules copy over to each stage, and which modules load. This section focuses on the following subset:

BOARD_VENDOR_RAMDISK_KERNEL_MODULES. This list of modules to be copied into the ramdisk.
BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD. This list of modules to be loaded in first stage init.
BOARD_VENDOR_RAMDISK_RECOVERY_KERNEL_MODULES_LOAD. This list of modules to be loaded when recovery or fastbootd is selected from the ramdisk.
BOARD_VENDOR_KERNEL_MODULES. This list of modules to be copied into the vendor or vendor_dlkm partition at /vendor/lib/modules/ directory.
BOARD_VENDOR_KERNEL_MODULES_LOAD. This list of modules to be loaded in second stage init.
The boot and recovery modules in ramdisk must also be copied to the vendor or vendor_dlkm partition at /vendor/lib/modules. Copying these modules to the vendor partition ensures the modules aren't invisible during second stage init, which is useful for debugging and collecting modinfo for bugreports.

The duplication should cost minimal space on the vendor or vendor_dlkm partition as long as the boot module set is minimized. Make sure that the vendor's modules.list file has a filtered list of modules in /vendor/lib/modules. The filtered list ensures boot times aren't affected by the modules loading again (which is an expensive process).

Ensure that recovery mode modules load as a group. Loading recovery mode modules can be done either in recovery mode, or at the beginning of the second stage init in each boot flow.

You can use the device Board.Config.mk files to perform these actions as seen in the following example:

```
# All kernel modules
KERNEL_MODULES := $(wildcard $(KERNEL_MODULE_DIR)/*.ko)
KERNEL_MODULES_LOAD := $(strip $(shell cat $(KERNEL_MODULE_DIR)/modules.load)

# First stage ramdisk modules
BOOT_KERNEL_MODULES_FILTER := $(foreach m,$(BOOT_KERNEL_MODULES),%/$(m))

# Recovery ramdisk modules
RECOVERY_KERNEL_MODULES_FILTER := $(foreach m,$(RECOVERY_KERNEL_MODULES),%/$(m))
BOARD_VENDOR_RAMDISK_KERNEL_MODULES += \
    $(filter $(BOOT_KERNEL_MODULES_FILTER) \
            $(RECOVERY_KERNEL_MODULES_FILTER),$(KERNEL_MODULES))

# ALL modules land in /vendor/lib/modules so they could be rmmod/insmod'd,
# and modules.list actually limits us to the ones we intend to load.
BOARD_VENDOR_KERNEL_MODULES := $(KERNEL_MODULES)
# To limit /vendor/lib/modules to just the ones loaded, use:
# BOARD_VENDOR_KERNEL_MODULES := $(filter-out \
#     $(BOOT_KERNEL_MODULES_FILTER),$(KERNEL_MODULES))

# Group set of /vendor/lib/modules loading order to recovery modules first,
# then remainder, subtracting both recovery and boot modules which are loaded
# already.
BOARD_VENDOR_KERNEL_MODULES_LOAD := \
        $(filter-out $(BOOT_KERNEL_MODULES_FILTER), \
        $(filter $(RECOVERY_KERNEL_MODULES_FILTER),$(KERNEL_MODULES_LOAD)))
BOARD_VENDOR_KERNEL_MODULES_LOAD += \
        $(filter-out $(BOOT_KERNEL_MODULES_FILTER) \
          $(RECOVERY_KERNEL_MODULES_FILTER),$(KERNEL_MODULES_LOAD))

# NB: Load order governed by modules.load and not by $(BOOT_KERNEL_MODULES)
BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD := \
        $(filter $(BOOT_KERNEL_MODULES_FILTER),$(KERNEL_MODULES_LOAD))

# Group set of /vendor/lib/modules loading order to boot modules first,
# then the remainder of recovery modules.
BOARD_VENDOR_RAMDISK_RECOVERY_KERNEL_MODULES_LOAD := \
    $(filter $(BOOT_KERNEL_MODULES_FILTER),$(KERNEL_MODULES_LOAD))
BOARD_VENDOR_RAMDISK_RECOVERY_KERNEL_MODULES_LOAD += \
    $(filter-out $(BOOT_KERNEL_MODULES_FILTER), \
    $(filter $(RECOVERY_KERNEL_MODULES_FILTER),$(KERNEL_MODULES_LOAD)))
```

This example showcases an easier-to-manage subset of BOOT_KERNEL_MODULES and RECOVERY_KERNEL_MODULES to be specified locally in the board configuration files. The preceding script finds and fills each of the subset modules from the selected available kernel modules, leaving the reamining modules for second stage init.

For second stage init, we recommend running the module loading as a service so it doesn't block boot flow. Use a shell script to manage the module loading so that other logistics, such as error handling and mitigation, or module load completion, can be reported back (or ignored) if necessary.

You can ignore a debug module load failure that isn't present on user builds. To ignore this failure, set the vendor.device.modules.ready property to trigger later stages of init rc scripting bootflow to continue onto the launch screen. Reference the following example script, if you have the following code in /vendor/etc/init.insmod.sh:

```
#!/vendor/bin/sh
. . .
if [ $# -eq 1 ]; then
  cfg_file=$1
else
  # Set property even if there is no insmod config
  # to unblock early-boot trigger
  setprop vendor.common.modules.ready
```

```
  setprop vendor.device.modules.ready
  exit 1
fi

if [ -f $cfg_file ]; then
  while IFS="|" read -r action arg
  do
    case $action in
      "insmod") insmod $arg ;;
      "setprop") setprop $arg 1 ;;
      "enable") echo 1 > $arg ;;
      "modprobe") modprobe -a -d /vendor/lib/modules $arg ;;
     . . .
    esac
  done < $cfg_file
fi
```

In the hardware rc file, the one shot service could be specified with:

```
service insmod-sh /vendor/etc/init.insmod.sh /vendor/etc/init.insmod.<hw>.cfg
    class main
    user root
    group root system
    Disabled
    oneshot
```

Additional optimizations can be made after modules move from the first to second stage. You can use the modprobe blocklist feature to split up the second stage boot flow to include deferred module loading of nonessential modules. Loading of modules used exclusively by a specific HAL can be deferred to load the modules only when the HAL is started.

To improve apparent boot times, you can specifically choose modules in the module loading service that are more conducive to loading after the launch screen. For example, you can explicitly late load the modules for video decoder or Wi-Fi after the init boot flow has been cleared (sys.boot_complete Android property signal, for example). Make sure the HALs for the late loading modules block long enough when the kernel drivers aren't present.

Alternatively, you can use init's wait<file>[<timeout>] command in the boot flow rc scripting to wait for select sysfs entries to show that driver modules have completed the probe operations. An example of this is waiting for the display driver to complete loading in the background of recovery or fastbootd, before presenting menu graphics.

Initialize the CPU frequency to a reasonable value in the bootloader
Not all SoCs/products might be able to boot the CPU at the highest frequency due to thermal or power concerns during boot loop tests. However, make sure the bootloader sets the frequency of all the online CPUs to as high as safely possible for a SoC or product. This is very important because, with a fully modular kernel, the init ramdisk decompression takes place before the CPUfreq driver can be loaded. So, if the CPU is left at the lower end of its frequency by the bootloader, the ramdisk decompression time can take longer than a statically compiled kernel (after adjusting for ramdisk size difference) because the CPU frequency would be very low when doing CPU intensive work (decompression). The same applies to memory and interconnect frequency.

Initialize CPU frequency of big CPUs in the bootloader
Before the CPUfreq driver is loaded, the kernel is unaware of the CPU frequencies and doesn't scale the CPU sched capacity for their current frequency. The kernel might migrate threads to the big CPU if the load is sufficiently high on the little CPU.

Make sure the big CPUs are at least as performant as the little CPUs for the frequency at which the bootloader leaves them in. For example, if the big CPU is 2x as performant as the little CPU for the same frequency, but the bootloader sets the little CPU's frequency to 1.5 GHz and the big CPU's frequency to 300 MHz, then the boot performance is going to drop if the kernel moves a thread to the big CPU. In this example, if it is safe to boot the big CPU at 750 MHz, you should do so even if you don't plan to explicitly use it.

Drivers shouldn't load firmware in first stage init
There might be some unavoidable cases where firmware needs to be loaded in first stage init. But in general, drivers shouldn't load any firmware in first stage init, especially in device probe context. Loading firmware in first stage init causes the entire boot process to stall if the firmware isn't available in the first stage ramdisk. And even if the firmware is present in the first stage ramdisk, it still causes an unnecessary delay.

Here's a **simple summary** of all the key **boot time optimization techniques** from that guide, broken down with the **why** and **how** in easy language:

🚀 1. Strip Debug Symbols from Kernel Modules

- **What?** Remove debug info (`.symtab`, `.strtab`) from `.ko` files.
- **Why?**
  - Speeds up flash read, ramdisk decompression, and module load.
  - Saves several **seconds** on boot.
- **How?**
  - Enable:

    ```make
    BOARD_DO_NOT_STRIP_VENDOR_RAMDISK_MODULES := false
    ```

  - Set in your device's `BoardConfig.mk`.

⚡ 2. Use LZ4 Compression (Instead of Gzip)

- **What?** Compress ramdisk and kernel with LZ4.
- **Why?** LZ4 decompresses **much faster**, even if it's slightly bigger.
- **Result:** Saves **0.5s to 1s** of boot time.
- **How?**
  - In `BoardConfig.mk`:

    ```make
    BOARD_RAMDISK_USE_LZ4 := true
    ```

  - Kernel: Use `CONFIG_KERNEL_GZIP=n` and `CONFIG_KERNEL_LZ4=y` in defconfig.

🧹 3. Reduce Logging in Drivers

- **What?** Avoid too many `printk()` or logging statements in `.ko` drivers.
- **Why?**
  - Each log adds a jump instruction → adds **RELA entries** → slows down module loading.
  - Some modules took 250ms just because of logs!
- **Tip:** Clean out logspam, especially in probe/init paths.

🔄 4. Enable Asynchronous Probing for Slow Drivers

- **What?** Let drivers load in the background (async) if they're slow.
- **Why?** Avoid blocking entire boot on slow drivers (I2C, firmware loaders, etc.).
- **Result:** Saves **100ms to 500ms**.
- **How?**
  - Add this in cmdline:

    ```ini
    mydriver.async_probe=1
    ```

  - Or pass it in modprobe config.

📈 5. Probe CPUfreq Driver Early

- **What?** Load CPU frequency scaling driver early.
- **Why?** The sooner CPUfreq loads, the sooner CPUs can go faster → faster boot.

- **How?**
  - Use **`MODULE_SOFTDEP()`** in driver to force early load.
  - Make sure required clocks, regulators, and thermal drivers load first.

📦 6. Move Non-Essential Modules to Second Stage Init

- **What?** Only keep **must-have** drivers (e.g., storage, watchdog) in ramdisk.
  Move the rest to **`/vendor/lib/modules`**.
- **Why?**
  - Smaller ramdisk → faster flash read and decompress.
  - Second-stage init runs in parallel → hides load time.
- **Result:** Saves **500ms to 1s** boot time.

📜 7. BoardConfig.mk Controls for Module Placement

Setting	Use
`BOARD_VENDOR_RAMDISK_KERNEL_MODULES`	Modules to **copy into** ramdisk
`BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD`	Modules to **load in 1st stage**
`BOARD_VENDOR_KERNEL_MODULES`	Modules for **`/vendor/lib/modules/`**
`BOARD_VENDOR_KERNEL_MODULES_LOAD`	Modules to load in 2nd stage init
`BOARD_VENDOR_RAMDISK_RECOVERY_KERNEL_MODULES_LOAD`	Modules needed for recovery

🧠 8. Module Loading Script (Service)

- Use **`/vendor/etc/init.insmod.sh`** as a **one-shot service** to load modules from **`/vendor/lib/modules`**.
- Report success/failure via properties like:

```sh
setprop vendor.device.modules.ready 1
```

- This lets **`init.rc`** continue to boot only after essential modules are ready.

📶 9. Late Load HAL-Specific Modules

- **What?** Load modules for e.g., Wi-Fi or decoder only when HAL starts.
- **How?**
  - Use **`modprobe blocklist`**
  - Load after boot complete (**`sys.boot_completed=1`**)
- **Why?** Reduces boot-time module clutter and delays.

🧮 10. Bootloader: Set CPU to High Frequency

- **Why?** Before CPUfreq driver loads, kernel uses whatever freq bootloader left.
- **Tip:** Set CPU (and memory/interconnect) freq high in bootloader to reduce:
  - Ramdisk decompression time
  - Init execution speed

🧱 11. Avoid Firmware Loads in First Stage Init

- **Why?**

  - If firmware isn't present in the ramdisk, it stalls boot.
- **Fix:**
  - Delay firmware-dependent drivers to second stage.
  - Or ensure firmware is pre-packed.

📊 TL;DR Optimizations Summary

Tip	Boot Time Saving
Strip debug symbols	1–2s
Use LZ4	0.5–1s
Reduce logging	100–300ms
Async probe	100–500ms
Early CPUfreq	Depends on SoC
Move to vendor partition	500ms–1s
Set CPU freq high in bootloader	Crucial for fast init
Avoid early firmware loads	Prevent boot stalls

Would you like a **checklist**, **example `BoardConfig.mk`**, or a **graph** of boot stages and module loading to visualize this better?

33. explain in simple words

Absolutely! Let's explain **Android boot time optimization** like you're helping a phone boot faster in real life. No technical jargon—just **simple, real-world analogies**.

🚀 Goal: Make the phone boot faster

When your phone turns on, it runs a bunch of steps:

1. **Reads the kernel and ramdisk from storage**
2. **Decompresses them (unzips them)**
3. **Loads drivers (like for touchscreen, Wi-Fi, USB, display...)**
4. **Starts Android system (home screen, etc.)**

Each step takes time. You can **shave seconds** off if you optimize things.

✅ Simple Ways to Make Boot Faster

1. Remove "debug info" from drivers

- **Imagine:** You packed a bag with things you don't need (like papers you won't read).
- Removing that **makes it lighter**.
- Same with drivers (modules). Remove debug info → they load faster.
- 🕒 Saves: 1 to 2 seconds

---

2. Use LZ4 (a fast unzip method)

- **Gzip** is like zip file: small but slow to unzip.
- **LZ4** is like a slightly bigger bag, but you open it faster.
- So the kernel and ramdisk unzip faster → boot is faster.
- 🕒 Saves: 0.5 to 1 second

---

3. Don't log too much during boot

- Every "log line" (like `printk("Driver started")`) takes time.
- Too much logging = too much delay.
- Reduce logs in drivers = load them faster.
- 🕒 Saves: 100–300 ms

---

4. Load slow drivers in the background

- Some devices take time to start (e.g., I2C sensors, firmware).
- Don't wait for them → load them **asynchronously** (in parallel).
- Phone doesn't pause just because 1 device is slow.
- 🕒 Saves: 100–500 ms

---

5. Start CPU speed control (CPUfreq) early

- Your CPU can go faster, but it needs a driver to control it.
- If that driver loads late, your CPU stays slow → everything else is slow.

- Load it early → everything becomes faster earlier.
- 🕒 Saves: small to large, depending on SoC

6. Move unimportant drivers to later stage

- Only load essential drivers first (e.g., storage, watchdog).
- Delay others (e.g., Wi-Fi, camera) to second stage.
- Smaller first boot stage = faster read + unzip + load.
- 🕒 Saves: 500 ms – 1 sec

7. Tell Android where drivers should load

- In your device config, say:
  - These drivers → first stage (must be in ramdisk)
  - These → second stage (put them in vendor partition)
- This keeps things organized and fast.

8. Make sure CPUs run fast early (bootloader job)

- Before the system loads, bootloader controls CPU speed.
- Set it to a high (but safe) speed so that unzipping and loading are faster.
- It's like starting the race with sprint mode on.

9. Don't load firmware too early

- Some drivers need firmware files (like Bluetooth).
- If it's missing in ramdisk = boot waits (bad!)
- Solution: Delay firmware drivers or move firmware to vendor partition.

📊 In Short: What You Should Do

Trick	What It Means	Why It Helps
Strip debug	Remove junk info from drivers	Smaller + faster load
Use LZ4	Use faster unzip	Faster boot
Reduce logs	Fewer **`printk()`** in drivers	Faster module load
Async load	Load slow drivers in background	No waiting
Early CPUfreq	Let CPU go fast early	Speeds up everything
Move drivers	Keep only needed ones in ramdisk	Smaller, faster first stage
Fast CPU at boot	Bootloader sets high freq	Faster unzip
Delay firmware	Don't load firmware too soon	Prevents waiting/stalls

Let me know if you'd like:

- ✅ a **checklist**
- 🛠️ a sample **`BoardConfig.mk`**
- 📈 a diagram of "first stage" vs "second stage" boot

I'll make it even easier!

34. Debug features

bookmark_border
The Linux kernel provides many mechanisms to diagnose issues including tracing, BPF, and profiling. Most of these techniques are available in the Generic Kernel Image (GKI) binary kernel. When developing products, there are many more tools that are integrated into the development or integration processes, some of which require kernel features that aren't part of GKI. As with most GKI features, we encourage you to work with upstream maintainers to add these features to Linux.

Note: Kernel debugging tools and techniques are beyond the scope of this document.
Downstream debug features in vendor modules
Debugging features are similar in scope to other kinds of features and must follow the same processes as described in Android-specific patches.

Intrusive downstream debug features
Some useful debug features require special configuration options that are disabled in GKI and therefore require a special build of the kernel. These features might be disabled due to performance penalties or unsuitability for production use, but are indispensable for debugging certain types of issues. Examples of such tools include sanitizers (KASAN, UBSAN), debugfs, and other partner-specific features and tools.

It's up to you to determine how you manage special debug-enabled kernels that impact the Kernel Module Interface (KMI). A common practice is to have an internal debug build and deliver the resulting artifacts to internal customers or downstream development partners. We recommend continuing to isolate such features into modules as partner-specific debug features can't be accepted into the GKI kernel beyond the guidelines described in Android-specific patches.

Here's a **simple explanation** of the debug features in Android kernels — especially in the context of **GKI (Generic Kernel Image)** and **vendor modules**:

🛠️ What Are Kernel Debug Features?

These are **tools built into the Linux kernel** to help developers **find bugs**, **trace issues**, or **profile performance**. Examples include:

- **Tracing** tools (e.g., `ftrace`, `tracepoints`)
- **Sanitizers** like:
  - **KASAN** (detect memory bugs)
  - **UBSAN** (detect undefined behavior)
- **BPF** tools (for profiling or monitoring)
- `debugfs` (file system for kernel debugging)

✅ What's Available in GKI by Default?

- The **Generic Kernel Image (GKI)** has a **minimal set** of debug features enabled.
- These are safe for production use and **won't hurt performance** too much.
- Examples: basic tracing, some BPF support, etc.

⚠️ What's Not Included in GKI?

Some **advanced or intrusive debugging features** are **disabled in GKI**, because:

- They **slow down** the system
- They may **affect stability**
- They're not needed in **production builds**

These include:

- `KASAN`, `UBSAN`, `debugfs`
- Partner-specific (custom) debugging tools

🧪 So What Do You Do If You Need These?

If you're debugging a serious issue (e.g., memory bug), you may need to:

- Build a **special debug version** of the kernel
- Enable those heavy debug options
- Use that build **only internally**

This debug build:

- Is **not for public use**
- Is often **not compatible** with the standard GKI module interface (KMI)

🧩 What About Vendor Modules?

- If your custom debug features are in **vendor modules** (your own `.ko` files), that's fine.
- But they still can't go into the **GKI** kernel — they remain **local to your project**.

📎 Good Practices

Tip	Why
Put debug features in modules	So they don't affect the core kernel
Keep a separate **debug build**	For internal testing only
Don't expect debug code to go upstream	It likely won't be accepted
Coordinate with partners (like Google)	If you want to upstream anything later

---

✅ Summary in Simple Words

Term	Meaning
GKI	The "Google-approved" shared kernel core
Debug features	Tools to find bugs (like memory issues or logs)
Intrusive debug features	Advanced tools that slow down the system
Vendor module	Your custom driver or debug tool in `.ko` file
Debug build	Special kernel build with debugging turned on

Would you like a **sample `.config` flags list** to enable/disable debug features in a kernel build?

35. yes

Here's a **sample `.config` list** showing common **debug features** in the Linux kernel used in Android development, with explanation and use-case for each. You can include these in your kernel `.config` file or use `make menuconfig` to enable them.

✅ Recommended Debug Features for Development Builds

Config Flag	Purpose	Why Enable It?
`CONFIG_KASAN=y`	**Kernel Address Sanitizer**	Detects out-of-bounds memory access (heap)
`CONFIG_UBSAN=y`	**Undefined Behavior Sanitizer**	Detects type/bounds/overflow issues
`CONFIG_DEBUG_INFO=y`	Build debug info (DWARF symbols)	Useful for crash reports, backtraces
`CONFIG_DEBUG_KERNEL=y`	Master switch for debugging	Enables other debug options
`CONFIG_DEBUG_FS=y`	Enables `/sys/kernel/debug/`	Exposes useful debug info
`CONFIG_PROC_KCORE=y`	Enables `/proc/kcore`	Used for kernel crash dumps
`CONFIG_FRAME_POINTER=y`	Keeps frame pointers	Better backtraces (needed for perf/ftrace)
`CONFIG_STACKTRACE=y`	Kernel stack traces	Allows printing call stacks
`CONFIG_MAGIC_SYSRQ=y`	Enable SysRq keys	Helps crash recovery/debugging via keyboard or adb
`CONFIG_FUNCTION_TRACER=y`	Core ftrace support	Used for kernel function tracing
`CONFIG_FTRACE=y`	Kernel tracing framework	See what the kernel is doing live
`CONFIG_BPF=y`	Enables BPF	Used for modern monitoring and tracing
`CONFIG_KPROBES=y`	Hook into kernel functions	Trace kernel behavior without recompiling
`CONFIG_MODULE_UNLOAD=y`	Allow unloading `.ko` modules	Lets you `rmmod` modules during debug

🔐 Disable These for Production

Config Flag	Why Disable for Production
`CONFIG_KASAN`	Large memory overhead (~2x RAM)
`CONFIG_UBSAN`	Slower performance
`CONFIG_DEBUG_FS`	May leak sensitive kernel data
`CONFIG_DEBUG_INFO`	Increases image size significantly
`CONFIG_MAGIC_SYSRQ`	Could expose control to attackers

🛠️ How to Add These

1. If you're building kernel manually:
    - Open defconfig or use `menuconfig`:

    ```bash
    make ARCH=arm64 menuconfig
    ```

2. Add directly to your defconfig (e.g. `arch/arm64/configs/vendor_defconfig`):

    ```bash
    CONFIG_KASAN=y
    CONFIG_DEBUG_INFO=y
    ```

3. Or append to `.config`:

    ```bash
    echo "CONFIG_KASAN=y" >> .config
    ```

4. Rebuild the kernel:

```bash
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-android- kernel_image
```

Would you like a **ready-made debug defconfig** file or a breakdown of how to load and test modules with debugfs or ftrace?

36. Kernel release notes

bookmark_border
This page summarizes the major features in each kernel release and provides links to additional information.

Kernel 6.12 new features
This section explains new features in kernel 6.12.

Memory allocation profiling
The key to understanding memory use is knowing where allocations are made. Kernel 6.12 includes a new memory allocation attribution system called memory allocation profiling (CONFIG_MEM_ALLOC_PROFILING in the config). With memory allocation profiling, each allocation is attributed to a unique source line so that issues with allocations can be identified quickly. Additionally, memory allocation profiling:

Is used during the engineering phase, but is available in the standard GKI image.

Can be enabled using the sysctl.vm.mem_profiling boot parameter.

Works both for in-kernel and loaded modules.

Faster io_uring with zero-copy and multishot read
In kernel 6.12, the statsd and logd modules use sendfile zero-copy, improving their performance.

Additionally, this kernel version implements multishot read where a single read operation can retrieve multiple pieces of data simultaneously, improving performance.

Improved Berkeley Packet Filter (BPF) capabilities and support
In kernel 6.12, the BPF toolchain has been moved to support CO-RE and several modern features. Additionally, a new BPF loader enables use of modern BPF for programs that are part of AOSP.

Proxy execution
Proxy execution lets the scheduler borrow CPU cycles from high-priority processes to recover locks held by lower-priority processes. This feature mitigates priority inversion problems.

Kernel 6.6 new features
This section explains new features in kernel 6.6.

Rust support
Multiple kernel 6.6 projects use Rust.

Per-virtual memory area (VMA) locks
Kernel 6.6 uses per-virtual memory area locks to address contention issues with mmap_sem (formerly known as mmap_lock). As such, apps that use a high number of threads might see launch times reduced by as much as 20%.

Earliest Eligible Virtual Deadline First (EEVDF) scheduler replaces CFS
The EEVDF replaces the Completely Fair Scheduler (CFS) to better balance CPU access between short and long-running tasks.

Reduced power consumption from read copy update (RCU) callbacks
The RCU_LAZY option uses a timer-based RCU callback batching method to save power. For a lightly loaded or idle system, this option can reduce the power consumed by 5% to 10%.

Better ZRAM memory compression
The new CONFIG_ZRAM_MULTI_COMP build setting lets ZRAM recompress pages with one of three alternative algorithms. This recompression further shrinks compressed memory, providing more free space for active tasks.

Kernel 6.1 new features
This section explains new features in kernel 6.1.

Faster security with kernel control flow integrity (KCFI)
KCFI replaces control flow integrity (CFI) resulting in a reduced runtime cost and no build-time cost. The reduced runtime cost allows KCFI to be enabled in more places compared to CFI, notably tracepoints and vendor hooks.

In addition to KCFI, kernel 6.1 introduces multiple security features, such as strict memcpy bounds checking and straight-line speculation attack mitigations.

For further information on KCFI, see Kernel control flow integrity.

Multi-Generational LRU (MGLRU)
MGLRU has been added to kernel 6.1 to improve memory management by better identifying which pages are actually in use. This improvement reduces the need for halting apps when the system experiences a memory shortage. This update also improves user experience as overall device responsiveness is better.

The implementation of MGLRU also includes support for a new RCU-safe maple tree, which, in some cases, can be used to replace red-black tree (rbtree). When used, the RCU-safe maple tree increases performance because of its lower footprint and locklessness.

For further information on MGLRU, see Multi-Gen LRU.

Scheduling
Maintaining and updating the scheduler is a key aspect to the work to improve the kernel. Kernel updates in 6.1 include:

Added cluster-aware scheduling, which increases performance by migrating to cores that share L2 cache.
Removed unnecessary energy-margin heuristic. By limiting some migrations, this update improves energy utilization up to 5%.
Improved load balancing to reduce wakeup latency.
Moved RCU expedited grace periods to a realtime kthread. This update greatly reduced RCU-related latency outliers.
Graphics
Kernel 6.1 contains new methods to dma-buf for exporting and importing sync files, aligning with the need for Vulcan Video API.

The new futex_waitv() method simplifies the porting of games from other platforms by waiting on multiple futexes simultaneously.

Debug tooling
Use the Kernel Concurrency SANitizer (KCSAN) in debug builds to identify races in kernel code.

Additionally, use the Kernel Memory SANitizer (KMSAN) to find uninitialized values in the kernel.

ARM64 support improvements
The 6.1 kernel brings multiple improvements for the ARM64 architecture, including:

Support for the ARMv8.6 timer extensions
Support for QARMA3 pointer-authentication algorithm
Initial support for ARMv9 Scalable Matrix Extension (SME)
Improvements to alternatives feature patching resulting in a smaller kernel image size

Here's a **simple summary of the Android kernel release notes** for versions **6.1**, **6.6**, and **6.12** — explained in **plain words** with examples and real impact for device developers.

🧠 Why This Matters

Each kernel version brings **new features** to:

- Improve performance 🏎️
- Save battery 🔋
- Boost security 🔐
- Make debugging easier 🛠️

🐧 Kernel 6.12 – Latest (2025+)

Feature	What It Means	Benefit
✅ **Memory allocation profiling** (`CONFIG_MEM_ALLOC_PROFILING`)	Tells you where memory is being used in kernel (line by line)	Find memory leaks easily
🚀 **Faster io_uring** (zero-copy & multishot)	Speeds up system calls (like log writing) using fewer copies	Faster logs, stats collection
🔄 **Improved BPF** (CO-RE support)	Modern BPF tools now work in AOSP directly	Advanced runtime tracing
🧵 **Proxy execution**	High-priority thread can "borrow" CPU from low-priority one to release a lock	Fixes **priority inversion bugs**

🧩 Kernel 6.6 – Introduced Big Changes

Feature	What It Means	Benefit
🦀 **Rust support**	You can now write kernel drivers in Rust	Safer driver development (less memory bugs)
🧠 **Per-VMA locks**	Each memory region has its own lock (vs global)	Faster app launches (esp. multi-threaded apps)
🧮 **EEVDF Scheduler**	New task scheduler replacing old "CFS"	Better CPU balance for all types of tasks
💡 **RCU Lazy Callbacks**	Defers cleanup work when idle	5–10% power savings during low usage
🧊 **Smarter ZRAM compression** (`CONFIG_ZRAM_MULTI_COMP`)	Tries 3 compression methods to shrink RAM usage more	More RAM space = better multitasking

🔐 Kernel 6.1 – Still Common in 2024–2025

Feature	What It Means	Benefit
🔐 **KCFI (Kernel Control Flow Integrity)**	Blocks some types of hacks by verifying function calls	More secure kernel, low overhead
🧠 **MGLRU (Multi-Gen LRU)**	Smarter RAM usage tracking	Fewer app kills, smoother UI under pressure
📉 **Better Scheduling**	Core-aware and energy-aware task placement	Improved performance + 5% better battery life
🎮 **futex_waitv()**	Waits on multiple futexes	Helps port PC/console games to Android
🎥 **dma-buf improvements**	Enhancements for Vulkan Video API	Better support for next-gen graphics/video
📏 **KCSAN & KMSAN**	Tools to find bugs in kernel (race conditions, uninitialized vars)	Better debugging during development
🔗 **ARM64 Improvements**	More hardware features supported (ARMv8.6, SME, QARMA3)	Supports newer ARM chips properly

🏁 Final Notes

If you're...	You should...
Building a new device	Use the **latest kernel** you can (e.g. 6.6 or 6.12)
Debugging memory bugs	Enable **KASAN/KMSAN** or **MEM_ALLOC_PROFILING**
Optimizing battery	Use 6.6+ for **RCU Lazy** and **MGLRU**
Shipping a product	Stick to **GKI-compliant features** only

Would you like:

- 🔧 A table of recommended `.config` flags for these features?
- 📈 A chart showing boot-time or battery gains per kernel?
- 📁 Android source tree file locations for each kernel component?

Let me know!

37. Stable kernel releases and updates

bookmark_border
The Linux kernel stable release model started in 2005, when it was determined that the existing kernel development model (a new release every 2-3 months) was not meeting the needs of most users. Users wanted bugfixes made during those 2-3 months, and Linux distributions found it difficult to keep kernels up to date without feedback from the kernel community. In general, attempts to keep individual kernels secure and with the latest bugfixes was a large and confusing effort by lots of different individuals.

Stable kernel releases are based directly on Linus Torvalds' releases, and are released every week or so, depending on various external factors (time of year, available patches, maintainer workload, etc.). The numbering of the stable releases starts with the number of the kernel release, and an additional number is added to the end of it. For example, the 4.4 kernel is released by Linus, and then the stable kernel releases based on this kernel are numbered 4.4.1, 4.4.2, 4.4.3, and so on. This sequence is usually shortened with the number 4.4.y when referring to a stable kernel release tree. Each stable kernel release tree is maintained by a single kernel developer, who is responsible for picking the needed patches for the release and managing the review/release process.

Stable kernels are maintained for the length of the current development cycle. After Linus releases a new kernel, the previous stable kernel release tree is stopped and users must move to the newer released kernel.

Long-term stable kernels
After a year of this new stable release process, it was determined that many different users of Linux wanted a kernel to be supported for longer than just a few months. In response, the Long Term Supported (LTS) kernel release was created, with the first LTS kernel (2.6.16) released in 2006. Since then, a new LTS kernel has been selected once a year and kernel community maintains that kernel for a minimum of 2 years.

At the time of this writing, the LTS kernels are the 4.4.y, 4.9.y, 4.14.y, 4.19.y, 5.4.y, and 5.10.y releases. A new kernel is released weekly. Due to the needs of some users and distributions, a few additional older kernels are maintained by kernel developers at a slower release cycle. Information about all long-term stable kernels, who is in charge of them, and how long they are maintained, can be found on the kernel.org releases page.

LTS kernel releases average 6-8 patches accepted per day, while the normal stable kernel releases contain 10-15 patches per day. The number of patches fluctuates per release given the current time of the corresponding development kernel release, and other external variables. The older a LTS kernel is, the less patches are applicable to it as many recent bugfixes are not relevant to older kernels. However, the older a kernel is, the harder it is to backport the changes that are needed to be applied, due to the changes in the codebase. So while there might be a lower number of overall patches being applied, the effort involved in maintaining a LTS kernel is greater than maintaining the normal stable kernel.

Stable kernel patch rules
The rules for what can be added to a stable kernel release have remained almost identical since its introduction and are summarized below:

Must be obviously correct and tested.
Must not be bigger than 100 lines.
Must fix only one thing.
Must fix something that has been reported to be an issue.
Can be a new device id or quirk for hardware, but not add major new functionality.
Must already be merged into Linus Torvalds' tree.
Note: For a full list of the rules for patches to be accepted into a stable kernel release, refer to the Documentation/process/stable_kernel_rules.rst kernel file.
The last rule, "Must already be merged into Linus Torvalds' tree", prevents the kernel community from losing fixes. The community never wants a fix to go into a stable kernel release that is not already in Linus Torvalds' tree, so that anyone who upgrades should never see a regression. This prevents many problems that other projects who maintain a stable and development branch can have.

Kernel updates
The Linux kernel community has promised its userbase that no upgrade ever breaks anything that is working in a previous release. That promise still holds true today. Regressions do happen, but those are the highest priority bugs and are either quickly fixed, or the change that caused the regression is quickly reverted from the Linux kernel tree.

This promise holds true for both the incremental stable kernel updates, as well as the larger major updates that happen every three months. However, the kernel community can only make this promise for the code that is merged into the Linux kernel tree. Any code that is merged into a device's kernel that is not in the kernel.org releases is unknown and interactions with it can never be planned for, or even considered.

Devices based on Linux that have large patch sets can have major issues when updating to newer kernels, because of the large number of changes between each release (10-14 thousand changes per release). SoC patchsets are especially known to have issues with updating to newer kernels due to their large size and heavy modification of architecture specific, and sometimes core, kernel code. As a result, most SoC vendors are starting to standardize on using the LTS releases for their devices, enabling those devices to receive bug and security updates directly from the Linux kernel community.

Security
When doing kernel releases, the Linux kernel community almost never declares specific changes as security fixes. This is due to the basic problem of the difficulty in determining if a bugfix is a security fix or not at the time of creation. Also, many bugfixes are only determined to be security related after much time has passed, so the kernel community strongly recommends always taking all bugfixes that are released.

Note: For details on Linus Torvalds' statement on security fixes, refer to the relevant email thread.
When security problems are reported to the kernel community, they are fixed as soon as possible and pushed out publicly to the development tree and the stable releases. As described above, the changes are almost never described as a "security fix", but rather look like any other bugfix for the kernel. This is done to allow affected parties the ability to update their systems before the reporter of the problem announces it.

For details on reporting security bugs to the kernel community to get them resolved and fixed as soon as possible, refer to Security bugs in The Linux kernel user's and administrator's guide at www.kernel.org.

Because security bugs are not announced to the public by the kernel team, CVE numbers for Linux kernel-related issues are usually released weeks, months, and sometimes years after the fix was merged into the stable and development branches.

Keep a secure system
When deploying a device that uses Linux, we strongly recommend that all LTS kernel updates be taken by the manufacturer and pushed out to their users after proper testing shows the update works well. This has several advantages:

Releases have been reviewed by the kernel developers as a whole, not in individual parts.
It's difficult to determine which patches fix "security" issues and which do not. Almost every LTS release contains at least one known security fix, and many yet "unknown."
If testing shows a problem, the kernel developer community reacts quickly to resolve the issue.
Attempts to filter out only the changes you run results in a kernel tree that is impossible to merge correctly with future upstream releases.

Here's a **simple and clear explanation** of what the "Stable Kernel Releases and Updates" section means — with examples and key takeaways for Android/device developers:

🐧 Why Do Stable Kernel Releases Exist?

Before 2005, Linux was releasing a **new kernel every 2–3 months**, but many users wanted **bug fixes earlier**. Also, it was hard for Android or Linux distro maintainers to keep up.

So Linux introduced:

- **Stable kernel releases** (every week)
- **Long Term Support (LTS)** kernels (maintained for 2–6 years)

🔁 What Is a Stable Kernel?

🧱 Example:

- Linus Torvalds releases **kernel 5.10**
- Then we get **5.10.1**, **5.10.2**, ... (weekly bugfix releases)
- This series is called: `5.10.y` (short for "5.10 stable branch")

✅ Maintained By:

- One kernel maintainer per version (like Greg Kroah-Hartman for LTS)
- Adds only **small, tested, safe bugfixes**

🛡️ What Is an LTS Kernel?

LTS = Long Term Supported
It's a **stable kernel** that is **officially maintained for years** — great for Android & embedded devices.

Example LTS Versions	Supported
4.4, 4.9, 4.14	Older Android
4.19, 5.4, 5.10	Android 10–12

Example LTS Versions	Supported
5.15, 6.1	Android 13+ and newer
6.6+	Future devices

🧪 What Patches Go Into Stable/LTS Kernels?

Strict rules for any patch to enter stable:

- Must fix something real (not just add new features)
- Must be small (under 100 lines)
- Must already be accepted into **mainline kernel**
- Must not break anything (maintain backward compatibility)

📌 That means:
If a bugfix is accepted into 6.6 mainline, only **then** it can be added to `6.6.2`, `6.6.3` (stable).

🔐 What About Security?

- Linux kernel **doesn't label patches as "security"** — because many bugfixes only later turn out to be security issues.
- **Every bugfix could be a security fix.**
- So: **You should always take all stable/LTS updates**, not just "CVE patches".

📌 CVEs (security IDs) might come **months after** a fix is released. That's why filtering patches is risky.

📦 Updating Android Devices (Why It's Hard)

Many Android/SoC vendors **add 1000s of custom patches** to Linux kernel.
This makes **upgrading to a new version painful**.

So now most Android devices:

- **Start from an LTS version** (e.g., 5.10)
- Use Google's **ACK** or **GKI** system
- Stay on that version and **apply weekly LTS updates**

📈 Why You Should Care

If you're a...	You should...
Android device maker	Base your kernel on a supported **LTS version**
Driver developer	Make sure your fix is accepted **upstream first**, then backported
Security team	Always take **full stable/LTS updates**, not just CVE patches
Performance engineer	Use newer kernels (6.x) for **better schedulers, power saving**
Platform integrator	Keep modules minimal and clean to ease **future merges**

📌 Final Takeaways

- **Stable kernels = weekly safe bugfixes** for each major version (e.g., 6.1.y)
- **LTS kernels = maintained for years**, used by Android
- **Never cherry-pick patches** — apply full updates
- **Security fixes are silent** — don't wait for a CVE
- Keeping your kernel up to date is the **best defense** and improves stability & performance

Would you like:

- 📆 A timeline of LTS versions vs Android releases?
- 📁 List of stable kernel locations in AOSP or GKI sources?
- 🧪 How to write a patch that can go into upstream and LTS?

Let me know!

38. difference between ACK and GKI in simple words

Sure! Here's the **difference between ACK and GKI in simple words**:

✅ ACK (Android Common Kernel)

What is it?
A kernel made by Google for Android devices by taking a standard Linux Long-Term Support (LTS) kernel and adding **Android-specific patches**.

Why it exists:
Because Android needs extra features (like binder, ashmem, etc.) that Linux doesn't have.

What's inside:

- Linux LTS kernel (like 5.10 or 6.1)
- Android patches (e.g., scheduler tweaks, memory handling)
- Can be **device-specific** and **monolithic** (everything built-in)

📁 Example in source:

```bash
kernel/common/
```

✅ GKI (Generic Kernel Image)

What is it?
A special kind of ACK — newer and more modular. It separates the **core kernel** (same for all devices) from **vendor-specific drivers**.

Why it exists:
To **reduce fragmentation** and make it easier to update kernels across devices.

What's inside:

- A **common core kernel** used by all Android devices
- Hardware drivers (WiFi, display, camera) moved to **modules**
- Follows a strict **Kernel Module Interface (KMI)**

📁 Example in source:

```swift
kernel/prebuilts/boot/kernel-5.10-gki/
```

📊 Summary Table

Feature	ACK	GKI
Based on	LTS kernel + Android patches	Same
Purpose	Android-specific kernel	Unified & modular kernel
Vendor drivers	Can be built-in	Must be separate modules
Fragmentation	High (many custom versions)	Low (shared core kernel)
Update difficulty	Hard (custom per SoC)	Easier (common base + modules)
Used from Android version	Android 8+	Android 11+ (officially from 12)
Boot partition	Whole kernel image	Generic kernel + modules
Maintained by	Google	Google

🧠 In Simple Analogy

Imagine Android kernel like a car:

- **ACK** is a car with everything built in for a specific customer (engine, seats, color, wheels all fixed).
- **GKI** is a common car chassis — then vendors plug in their custom seats, wheels, engine modules on top.

This modular approach makes it easier to update the core (GKI) without rebuilding the entire car (ACK).

Would you like a diagram to show this visually?

39. GKI development

bookmark_border
This guide provides a high-level overview of GKI development.

Review existing documentation
Before you begin GKI development, read the following documents:

For an overview of the GKI project, read GKI project.

For an explanation of how Android Common Kernels (ACKs), GKI kernels, and KMI relate, read Android Common Kernels (ACK).

To learn how to contribute to the kernel, read Developing kernel code for GKI, Contribution guidelines for Android Common Kernels, and Linux kernel coding standards.

To learn about GKI and vendor modules, read Kernel modules overview and other documents in the modules section.

Build, monitor, and test
When building, monitoring, and testing your GKI implementations, follow these guidelines:

Use the hermetic build process and LLVM toolchain.

Submit your KMI symbol list to the appropriate ACK.

Set up a continuous ABI monitoring system in your internal kernel development workflows. For additional details, see Maintaining a Stable KMI.

Run full system validation tests with GKI. If you experience any regressions in power and performance and don't have mechanisms to address these issues in the ACK code, email kernel-team@android.com for help.

Follow the patch process
Before creating and sending a patch, send patches upstream to LKML. If the patch can't be upstreamed first, explain the rationale in a bug. For further information on patching, see Develop kernel code for GKI.

How to file a kernel bug or contact us
To report a bug, click the Kernel (GKI) File a bug link on the Reporting bugs page.

If you have GKI or kernel-related questions, email the kernel-team@android.com.

Here's a **simple explanation of the GKI development process** for Android kernel developers:

🧰 What is GKI Development?

GKI (Generic Kernel Image) development is about building, testing, and maintaining the **common Android kernel** that works across multiple devices, separating **core kernel logic** from **vendor-specific drivers**.

📋 Steps in GKI Development (in simple words):

1. ✅ Review Documentation First

Before you start, read the official GKI and ACK docs:

- **What is GKI & ACK?** → Understand the modular design.
- **How to write kernel code?** → Follow proper coding style and standards.
- **How to manage modules?** → Learn how vendor drivers work as `.ko` files.

---

2. 🔧 Build GKI Using Google's Toolchain

Use:

- The **hermetic build system** (completely self-contained).
- **LLVM compiler** (not GCC).

These ensure compatibility and reproducibility across Android devices.

---

3. 📈 Monitor KMI (Kernel Module Interface)

- You must **not break KMI** accidentally.
- Set up **automated checks** to detect changes in exported kernel symbols.
- If your driver depends on new kernel symbols, submit your symbol list to ACK.

📘 *KMI = the set of functions/variables that vendor modules can depend on.*

---

4. 🧪 Test Your GKI

- Use **full system testing** on real devices.
- Watch for **performance regressions**, crashes, or power issues.
- If you can't fix something yourself, contact:
  📧 `kernel-team@android.com`

---

5. 📤 Submit Patches Properly

- Try to **upstream** your kernel changes to the Linux Kernel Mailing List (LKML).
- If not possible (e.g., Android-only feature), explain **why** in a bug report.

All patches must follow:

- Coding style
- Review process
- KMI stability rules

---

6. 🐞 Report Bugs or Contact Help

If you hit problems:

- Go to the Android Reporting Bugs page.
- Choose **"Kernel (GKI)"** category.
- Or email 📧 `kernel-team@android.com`

✅ Example

Say you wrote a new touchscreen driver for your device:

- Build it as a **vendor module** (not part of GKI).
- Only use kernel symbols allowed by the **KMI symbol list**.
- Ensure it compiles with the **LLVM toolchain**.
- Test the full boot and touch functionality.
- Submit the patch with proper logs and justification.

🧠 Summary

Step	Action
1	Read GKI docs and coding rules
2	Build with LLVM and hermetic system

Step	Action
3	Track ABI/KMI changes continuously
4	Test on real devices
5	Submit patches with justification
6	Report issues or ask for help

Would you like a diagram to represent this flow visually?

40. explain in simple words

Sure! Let me explain **GKI (Generic Kernel Image) development** in very simple words, step-by-step:

🧠 What is GKI?

GKI is a **common Android kernel** made by Google that works on many phones. It has:

- **Core kernel** (same for all phones)
- **Vendor modules** (custom drivers for each phone)

You work on **vendor modules**, not the core.

🚶 Steps to Do GKI Development

---

1. Read Docs First

Before starting, you should read:

- What is GKI?
- How GKI talks to vendor drivers?
- How to write kernel code?

This helps you avoid mistakes.

---

2. Build the Kernel

Use Google's special tools:

- **Hermetic build** → Controlled environment
- **LLVM compiler** → Google's recommended compiler

Don't use GCC or old tools.

---

3. Check for Interface Breaks (KMI)

GKI and vendor modules talk through a "contract" called **KMI**.

You must not break this contract (like removing or changing a shared function).

So, set up **automatic checks** to detect any change.

---

4. Test Everything

After building:

- Run your kernel on a real device
- Make sure phone boots, performance is okay, and battery is fine

If there's a problem and you can't fix it, **email Google's kernel team**.

---

5. Submit Code

If you want to add new code:

- Try to send it to the **main Linux kernel** (upstream)
- If not possible, explain why (maybe it's Android-specific)

Follow Google's **code rules and style**.

6. Report Bugs or Get Help

If something breaks or you're stuck:

- Report a bug on Google's site
- Or send email to `kernel-team@android.com`

🧑‍🔧 Example

You write a new **Wi-Fi driver**:

- Build it as a **separate module (.ko)**
- Test it with the GKI kernel
- Make sure it doesn't break GKI's rules
- Submit the driver and explain how it works

🧾 Summary Table

Step	What You Do
1	Read official GKI docs
2	Build with LLVM & hermetic build system
3	Don't break the GKI-to-driver interface (KMI)
4	Test it on devices
5	Submit changes properly
6	Report issues if needed

Would you like me to show this as a **visual diagram**?

41. ↪
LLVM & hermetic
explain in details in simpl words

Sure, let's explain **"LLVM"** and **"hermetic build"** in **simple words**, with full detail and examples:

🔧 1. What is LLVM?

🟢 Simple Explanation:

LLVM is a **compiler framework** used to build programs — in our case, the **Android kernel**.

💡 Compare with GCC:

Normally, people used **GCC** (GNU Compiler Collection) to compile the Linux kernel.
But Google prefers **LLVM/Clang** for Android because:

Feature	GCC	LLVM/Clang (Used by Google)
Speed	Slower builds	Faster builds
Diagnostics	Fewer warnings	Better error messages
Compatibility	Works well	Works with Android features
Licensing	GPL	More flexible (BSD-like)

✅ Why Google uses LLVM for Android:

- Easy to integrate into Android build system
- Produces smaller & optimized kernel binaries
- Works well with Android-specific debug tools
- Google maintains Android patches for LLVM

👷 Example:

When you compile the kernel:

```bash
make LLVM=1
```

That tells the system:
🗣️ “Hey, build this kernel using LLVM instead of GCC.”

🔒 2. What is Hermetic Build?

🟢 Simple Explanation:

A **hermetic build** is a build that:

- Doesn’t depend on your system’s installed tools
- Always uses **predefined tool versions**
- Is **fully reproducible**

It’s like cooking using **a packaged kitchen** with **exact ingredients** — not your home kitchen where versions and tools may vary.

✅ Why Google uses Hermetic Build:

- Builds are **consistent** across all systems
- No surprise bugs due to local compiler differences
- Easier for Google to test, verify, and certify GKI

👷 Example:

You don't use **`/usr/bin/gcc`** or system-wide **`make`** — instead:

```bash
tools/bazel build //kernel:kernel_aarch64
```

Or

```bash
build/build.sh
```

This script:

- Pulls LLVM, scripts, headers, and configs from Android source
- Builds kernel in a sandboxed, clean environment

✅ Why LLVM + Hermetic Are Used Together?

Because together they:

- Ensure stable, repeatable, and **trusted** kernel builds
- Prevent issues due to environment differences
- Make Google's **Generic Kernel Image (GKI)** builds predictable

🧾 Summary

Term	Meaning (Simple)
LLVM	A modern compiler Google uses to build Android kernels
Hermetic Build	A clean, sandboxed, environment-controlled build system
Why used	To ensure reliable, same-everywhere kernel builds and faster development

Would you like a diagram to visualize how hermetic + LLVM fits into GKI build?

42. ↪
Hermetic
confusing

No problem! Let's break down **"hermetic build"** in **very simple words**, step by step:

🧱 What is a Hermetic Build?

Imagine you want to **bake a cake**.

🍰 Scenario 1: Non-hermetic build (Not good)

You bake it in your kitchen using:

- Whatever ingredients you have
- Whatever oven you have
- Whatever measuring tools you have

📌 Problem:
Your friend tries the same recipe at their home, but the cake comes out **different** because:

- Their oven is different
- They used different flour
- Their scale is not accurate

🍰 Scenario 2: Hermetic build (Good)

You give your friend a **cake-making kit** that has:

- Same ingredients
- Same oven
- Same tools
- Same steps

📌 Result:
Now the cake is **exactly the same** every time, anywhere, for everyone.

🛠️ In software (Android kernel build):

- A **non-hermetic build** depends on:
  - Your computer's installed compiler (e.g. GCC)
  - Your system's files and tools
  - Your environment variables
    🔴 This can break or produce different results.
- A **hermetic build**:
  - Uses a fixed version of all tools (e.g. Google's Clang/LLVM)
  - Runs in an isolated sandbox
  - Doesn't rely on your computer setup
    ✅ This guarantees **same output always**.

✅ Why Hermetic Builds are Useful in Android

Benefit	Explanation
🔁 Reproducible	Everyone gets **same kernel image**, no matter where it's built
🛠️ Debuggable	Bugs are easier to track, since builds are identical
📦 Reliable for devices	Ensures Android phones boot with trusted, verified kernels
🔒 Secure	Avoids accidental usage of bad tools or files

🧾 Simple Summary:

Term	Meaning
Hermetic Build	A **clean and fixed** way to build software using **known tools only**
Like what?	Like a **ready-made baking kit** that gives **same cake everywhere**

Would you like me to show how this works in a **kernel diagram** or in **source tree structure**?

43. so clang will be included here in hermetic?

Yes, exactly! ✅

In a **hermetic build**, the **Clang/LLVM toolchain is bundled inside** the build system. That means:

✅ Clang is included and fixed in hermetic build

- **No need to install Clang on your system** manually
- The **exact version** of Clang (and related tools like `ld.lld`, `llvm-objcopy`, etc.) is **predefined**
- Everyone building the kernel uses the **same Clang version**, no matter what's installed on their computer

📦 Where is it included?

In Android's case:

- A prebuilt Clang is provided in the **AOSP toolchain**.
- It's downloaded automatically or pre-synced with AOSP build (`prebuilts/clang/host/linux-x86/clang-r<version>`).
- The **Android build system** uses that Clang for building GKI, kernel modules, etc.

🔒 Why is this important?

Problem (non-hermetic)	Hermetic build solves it by
Developer A uses Clang 12, Developer B uses 15	Everyone uses the same Clang from `prebuilts/`
Different behavior or bugs due to tool version	Exact same output every time
Difficult to debug or reproduce build failures	Reproducible builds, easier debugging

💡 Summary:

> **Yes, Clang is included** in a hermetic kernel build.
> It's not taken from your local system.
> Instead, it's a **Google-provided prebuilt Clang** that ensures reproducibility and consistency.

Would you like to see how to enable or configure this in the Android build system (e.g., via `BUILD_CONFIG`, `ToolchainPath`, etc.)?

44. GKI versioning scheme

bookmark_border
This page describes the versioning scheme for Generic Kernel Images (GKIs). A Generic Kernel Image (GKI) has a unique identifier called the kernel release. The kernel release consists of the kernel module interface (KMI) version and the sublevel. The kernel release is specific to the image being released, whereas the KMI version represents the interface that a release is built from. A KMI version can support multiple kernel releases. A kernel release is tied to only one KMI version. In the unlikely event where the kernel module interface has to be changed, the KMI generation is iterated to reflect the change in KMI version.

Summary of terms
The following table summarizes important terms used on this page and for GKI updates.

Name Symbol Example Description
Kernel release w.x.y-zzz-k-suffix 5.4.42-android12-0-foo Unique identifier for a GKI release. This is the value returned by uname.
KMI version w.x-zzz-k 5.4-android12-0 Describes the kernel module interface (KMI) between GKI and dynamically loadable kernel modules (DLKM).
Sublevel y 42 Describes the release order of kernel releases within the same KMI version.
The following table lists other related terms as a reference.

Name Symbol Example Description
w.x.y w.x.y 5.4.42
For details, see Linux Kernel Makefiles (search for "KERNELRELEASE").

w.x.y is used directly throughout this document. This is also commonly referred to as the three-part version number. The term used in VINTF, kernel version, might cause confusion with other terms, especially w.

This variable is referred to as kernel_version_tuple in libkver.

This tuple must not be decreased by any updates, including OTA or mainline.

Kernel branch zzz-w.x android12-5.4 This term is used in Common kernel branch types.
Version w 5 This term is not used in this document. This variable is referred to as version in libkver.
Patch level x 4 This term is not used in this document. This variable is referred to as patch_level in libkver.
Android release zzz android12
This is the Android (dessert) release number that the kernel is associated with.

When comparing the AndroidRelease field, the numeric part is extracted from the string for comparison.

The Android release number must not be decreased by any updates, including OTA or mainline.

KMI generation k 0
This is an additional number added to deal with unlikely events. If a security bug fix requires changes to the KMI within the same Android release, a KMI generation is increased.

The KMI generation number starts with 0.

Versioning design
Kernel release
Definition
For devices that ship with GKI, the kernel release is defined as follows:

```
KernelRelease :=
Version.PatchLevel.SubLevel-AndroidRelease-KmiGeneration-suffix
w      .x        .y      -zzz          -k            -something
```

For more information, see Determining kernel release from a device.

The following is an example of a kernel release.

```
5.4.42-android12-0-00544-ged21d463f856
```

Description
The kernel release is the unique ID of a GKI release. If two GKI binaries have the same kernel release, they must be byte-wise identical.

A kernel release consists of a KMI version, a sublevel, and a suffix. For the purposes of this document, the suffix after KMI generation is ignored.

KMI version
Definition
The KMI version is defined as follows:

```
KmiVersion :=
Version.PatchLevel-AndroidRelease-KmiGeneration
w      .x        -zzz           -k
```

Note that the sublevel, y isn't part of the KMI version. For the example in Kernel release, the KMI version is:

```
5.4-android12-0
```

Description
The KMI version describes the kernel module interface (KMI) between GKI and dynamically loadable kernel modules (DLKM).

If two kernel releases have the same KMI version, they implement the same kernel module interface. The DLKMs that are compatible with one are also compatible with the other.

The KMI version must not be decreased by any OTA updates.

Sublevel
The sublevel, y, describes the release order of kernel releases within the same KMI version.

For two kernel releases that have the same KMI version but have sublevel Y1 and Y2 respectively:

If Y1 is less than or equal to Y2, a device running Y1 can receive an update to Y2.
If Y1 is greater than Y2, a device running Y1 can't be updated to Y2.
That is, if the KMI version doesn't change, the sublevel must not be decreased by any OTA update.

Determine kernel release from a device
The full kernel release can be found by executing uname -r, or uname(2) with the following code snippet:

```
std::string get_kernel_release() {
  struct utsname buf;
  return uname(&buf) == 0 ? buf.release : "";
}
```

An example output is:

```
5.4.42-android12-0-00544-ged21d463f856
```

For the purpose of this document, anything after the KMI generation is ignored when extracting kernel information. More formally, the output of uname -r is parsed with the following regex (assuming zzz always starts with "android"):

```
^(?P<w>\d+)[.](?P<x>\d+)[.](?P<y>\d+)-(?P<z>android\d+)-(?P<k>\d+).*$
```

The ignored information can include information such as the ci.android.com build number, number of patches on top of the baseline kernel, and SHA hashes of the git commit.

libkver
The library, libkver, provides a C++ interface to parse the kernel release or a KMI version string. For a list of APIs that libkver exposes, see packages/modules/Gki/libkver/include/kver.

VINTF checks
For Android 11 or lower, the Android release part of the KMI version is specified manually in the device manifest by device manufacturers. For details, see VINTF kernel match rules.

From Android S, the Android release part of the KMI version can be extracted from the kernel and injected into the device manifest at build time.

Because kernel configuration requirements generally don't change, there's no need to encode k within the compatibility matrix. However, in the unlikely case where the kernel configuration requirement does need to be changed, ensure the following:

The corresponding requirement from the compatibility matrix is removed.

Additional VTS tests are added to check the new requirements conditional on KMI generation.
Boot image version in OTA metadata
Even if the boot image is updated through OTA an update, it must be wrapped in the OTA payload format, payload.bin. The OTA payload encodes a version field for each partition. When update_engine handles an OTA payload, it compares this field to ensure the partition is not downgraded.

To avoid confusion, the version field for the boot partition in the OTA metadata is called boot image version.

Because the ramdisk is always built from scratch, using the ramdisk timestamp is sufficient to describe the whole boot image. There's no need to encode kernel release in the boot image version, unless you are stitching an old boot image to a new kernel binary in the future.

Before an OTA update, the OTA client checks the boot image version in the same way as any other partition.

Here's a **simplified explanation** of the **GKI versioning scheme** with examples:

🧠 What is GKI versioning?

Google uses a **structured naming scheme** to version every GKI (Generic Kernel Image) release. This helps:

- Match kernel images with the right modules
- Ensure updates don't break compatibility
- Track kernel updates in OTA (over-the-air) updates

📌 Key Components of the GKI Version

A GKI kernel version looks like this:

```
5.4.42-android12-0-00544-ged21d463f856
```

Let's break it down:

Part	Example	Meaning
5	Major version	Kernel version from Linux (e.g., 5.x.x)
4	Patch level	From upstream (e.g., 5.4.x)
42	Sublevel	Update number within the same KMI version
`android12`	Android release	Which Android release this kernel is for (e.g., android12 = Android 12)
`0`	KMI generation	Interface version number — only changes if kernel ABI must change
`00544-ged21d463f856`	Suffix (optional info)	Git SHA, build number (ignored when comparing compatibility)

💡 Terms Explained Simply

🟦 Kernel Release: Full version string from `uname -r`

```bash
uname -r
# Output: 5.4.42-android12-0-00544-ged21d463f856
```

🟧 KMI Version: Only this part:

```
5.4-android12-0
```

- Tells us: This kernel follows the same **module interface** rules.
- All kernels with the same KMI can load the **same `.ko` modules**.

🟨 Sublevel (42):

- Just means it's the 42nd patch release of this KMI version.

- **Used to track OTA version order**.
- Rule: OTA update must only go **forward**, not backward.

🔁 Example Use Case

Imagine you have two kernel releases:

1. `5.4.42-android12-0-...`
2. `5.4.45-android12-0-...`

- They have the **same KMI version**, so **same `.ko` modules** work.
- Sublevel increased from `42 → 45`, so it's a **valid OTA update**.
- If it were `45 → 42`, **OTA would reject it** as a downgrade.

✅ Summary (Simple)

Concept	What it means in real life
GKI version	A special name for the kernel build used in Android devices
KMI version	A compatibility label for kernel modules. If KMI matches, **`.ko`** files will load fine
Sublevel	A number showing update order. Must always go up in OTA updates
Suffix	Build metadata (Git SHA, etc.) — **not used** to check compatibility

---

Want a visual flow chart of GKI versioning or a quick cheat-sheet diagram?

45. Generic Kernel Image (GKI) release builds

bookmark_border
This page provides a listing of different versions of GKI release builds.

Execute the following to sync the kernel source code:

repo init -u https://android.googlesource.com/kernel/manifest
mv <kernel_manifest.xml> .repo/manifests
repo init -m manifest.xml
repo sync
Releases
Refer to the following for information on specific release builds:

android16-6.12 Releases
android15-6.6 Releases
android14-6.1 Releases
android14-5.15 Releases
android13-5.15 Releases
android13-5.10 Releases
android12-5.10 Releases

Here's a **simple explanation** of **Generic Kernel Image (GKI) release builds**:

✅ What is a GKI Release Build?

A **GKI release build** is an **official prebuilt Android kernel** made by Google that:

- Follows the **Generic Kernel Image** standard.
- Works across multiple Android devices and vendors.
- Is built from a specific **Android + Linux kernel version** combination.

📦 What's in a GKI Release?

Each GKI build includes:

- The **Linux kernel image** (`Image` or `Image.gz`)
- The **kernel modules** (`.ko` files)
- A **kernel ABI symbol list** (to ensure binary compatibility)
- Build metadata like version numbers and Git commit hashes

📋 Available GKI Versions

Each version is tied to an **Android release** and a **Linux kernel version**. Examples:

Android Version	Kernel Version	Directory Name
Android 16	Linux 6.12	`android16-6.12`
Android 15	Linux 6.6	`android15-6.6`
Android 14	Linux 6.1	`android14-6.1`
Android 14	Linux 5.15	`android14-5.15`
Android 13	Linux 5.15	`android13-5.15`
Android 13	Linux 5.10	`android13-5.10`
Android 12	Linux 5.10	`android12-5.10`

🔧 How to Sync GKI Source Code

To fetch GKI source code for a specific release:

```bash
repo init -u https://android.googlesource.com/kernel/manifest
mv <kernel_manifest.xml> .repo/manifests         # rename your manifest
repo init -m manifest.xml                        # use the renamed manifest
repo sync                                        # fetch all kernel source
```

> Replace `<kernel_manifest.xml>` with the actual manifest for the release you want (e.g., `android14-6.1.xml`).

💡 Why Use GKI Builds?

- Ensures **kernel module compatibility** (KMI compliance).
- Simplifies vendor integration and testing.
- Receives **security and performance updates** from Google.

Would you like help navigating to one of these versions or downloading a build?

46. ↪
🔧 How to Sync GKI Source Code To fetch GKI source code for a specific release: bash CopyEdit `repo init -u https://android.googlesource.com/kernel/manifest mv <kernel_manifest.xml> .repo/manifests # rename your manifest repo init -m manifest.xml # use the renamed manifest repo sync # fetch all kernel source`
make it simple

Sure! Here's a **very simple explanation** of how to sync GKI source code:

🔧 How to Download GKI Source Code

1. **Start repo:**

```bash
repo init -u https://android.googlesource.com/kernel/manifest
```

2. **Put the GKI manifest file in the right place:**

Download or copy a GKI manifest file (like **`android14-6.1.xml`**), and move it:

```bash
mv android14-6.1.xml .repo/manifests/
```

3. **Tell repo to use that manifest:**

```bash
repo init -m android14-6.1.xml
```

4. **Download the full kernel source:**

```bash
repo sync
```

✅ Now you have the GKI kernel source code for that version.

Let me know if you want help picking a version (like Android 14 with Linux 6.1).

47. why manifest here?

Great question! Here's a **simple explanation** of why the `manifest.xml` is used in GKI (and Android in general):

📦 Why is `manifest.xml` used when downloading GKI?

When using `repo` to download source code (like GKI or AOSP), you're not pulling from just **one** Git repository — you're pulling from **many repos** (kernel, toolchains, modules, etc.).

🔹 What does `manifest.xml` do?

It tells `repo` **which repositories to fetch**, **which branches to use**, and **where to put them** in your local directory.

Think of it like a **shopping list** 📋:

- Instead of downloading the entire supermarket,
- You only download the exact items you need for the GKI build.

🧠 Example

The GKI `manifest.xml` might say:

```xml
<project name="kernel/common" path="common" revision="android14-6.1" />
<project name="kernel/build" path="build/kernel" revision="android14" />
```

This means:

- Get the Linux kernel from `kernel/common` (Android 14 version).
- Get the kernel build scripts from `kernel/build`.

🛠️ Why not just use Git?

Because `git` works with only **one repo at a time**. But `repo` + `manifest.xml` can:

- Handle **multiple Git repos**
- Ensure they're all on the **correct version**
- Put them in the **correct folders**

Let me know if you want to see what's inside one of the actual GKI manifest files!

48. Generic Kernel Image (GKI) release process

bookmark_border
This page describes how GKI is released, including weekly, quarterly, and out of band emergency releases. The goal of this document is to give OEMs a guideline on where to pick up the GKI as well as the process for out of band emergency fixes. OEMs can also use GKI development to learn more about how they can work with the Android Kernel team to optimize the GKI kernel for their products.

GKI release cadence
GKI is released on a quarterly cadence post KMI freeze.

Release Month a12-5.10 a13-5.10 a13-5.15 a14-5.15 a14-6.1 a15-6.6 a16-6.12
June
2025 Check-in cut off
GKI preload ready Jun 16
Jun 30 Jun 2
Jun 16 Jun 2
Jun 16 Jun 2
Jun 18
July
2025 Check-in cut off
GKI preload ready Jul 16
Jul 31 Jul 16
Jul 31 Jul 16
Jul 31 Jul 1
Jul 15 Jul 1
Jul 15 Jul 1
Jul 15
August
2025 Check-in cut off
GKI preload ready Aug 1
Aug 15 Aug 1
Aug 15 Aug 1
Aug 15
September
2025 Check-in cut off
GKI preload ready Sep 16*
Sep 30* Sep 16
Sep 30 Sep 1
Sep 15 Sep 1
Sep 15 Sep 1
Sep 15
October
2025 Check-in cut off
GKI preload ready Oct 16
Oct 31 Oct 1
Oct 15 Oct 1
Oct 15
November
2025 Check-in cut off
GKI preload ready
December
2025 Check-in cut off
GKI preload ready Dec 1
Dec 15 Dec 1*
Dec 15* Dec 1
Dec 15 Dec 1
Dec 15
Note: Dates are subject to change.
GKI build validity for OEMs
OEMs can use a recently released Android GKI. OEMs can launch with GKI-certified builds as long as they are compliant with LTS requirements in the Android Security Bulletin (ASB).

Weekly development releases

Releases are tested with Cuttlefish to ensure they pass a minimum quality bar.
GKI binaries are available for self-service from Android CI as changes are merged. Weekly builds won't be certified, though can be used as a baseline for development and testing. Weekly builds can't be used for production device builds for end users.

Here's a **simple explanation** of the **GKI release process**:

🧠 What Is This About?

This is about how Google releases the **Generic Kernel Image (GKI)** over time, and how **OEMs (device makers)** like Samsung or Xiaomi can **use it in their devices**.

📅 1. GKI Release Cadence (Schedule)

- GKI releases happen **quarterly** (every 3 months), after something called the **KMI freeze** (when the kernel interface is locked).
- Example of GKI branches:
  - **`a14-5.15`**: Android 14 with Linux 5.15
  - **`a16-6.12`**: Android 16 with Linux 6.12

Example Timeline:

Month	Event
June	Code freeze, GKI ready on June 16 or June 30
September	Another GKI release
December	Yet another quarterly release

You'll see entries like:

> **Jul 1 – GKI preload ready**

That means: the GKI build for that branch is complete and ready for OEMs to pick up and integrate into their phones.

🛠️ 2. GKI Weekly Development Releases

- Google also pushes **weekly GKI builds** for internal testing.
- These builds:
  - ✅ Pass minimal tests (using virtual devices like **Cuttlefish**).
  - ❌ Are **not certified** for production.
  - 🔧 Can be used by OEMs for testing or development.

🏁 3. Using GKI for Your Product

- **OEMs can ship devices** using a **quarterly certified GKI** as long as:
  - It's from the proper Android version (e.g., **`android14-5.15`**).
  - It's updated with **security patches** (from ASB – Android Security Bulletins).
- OEMs **shouldn't ship weekly builds** to users — only use them for internal development.

🧯 4. Emergency (Out-of-Band) GKI Releases

If there's a critical bug (like a **security vulnerability or crash**), Google can do an **emergency GKI release**, even outside the normal schedule.

📦 Where do OEMs get GKI builds?

From **Android CI (Continuous Integration)** — Google hosts GKI builds here for download by OEMs.

Let me know if you want a visual timeline of the release cycle or help understanding terms like KMI, Cuttlefish, or ASB.

49. ↪
Where do OEMs get GKI builds?
explain more

Sure! Let's **explain in more detail** how and **where OEMs get GKI builds**, and what they do with them.

🔍 What is a GKI build?

A **GKI (Generic Kernel Image)** is a version of the Linux kernel built by Google that:

- ✅ Works across many Android devices.
- 🔒 Has a stable **KMI (Kernel Module Interface)**.
- 🛠️ Lets OEMs load their own hardware drivers as separate modules.

📦 Where do OEMs get GKI builds?

OEMs (like Samsung, Xiaomi, etc.) get GKI builds from **Android's public CI system (Continuous Integration)**. This is where Google automatically builds and tests kernels.

There are two main sources:

1. Quarterly Certified Releases (for production)

- ✅ These are stable and fully tested.
- 🧪 Run through full validation (CTS, VTS, etc.).
- 📥 Available from **GKI release repositories**.
- 📋 Each GKI build has a unique version like:

```
5.4.42-android12-0-00544-ged21d463f856
```

- 🔐 OEMs are allowed to ship these in commercial phones and tablets **if** they also follow Android Security Bulletin (ASB) updates.

📍 These are the **official** GKI builds used in real-world devices.

---

2. Weekly Development Builds (for testing only)

- ⚠️ Not certified for shipping.
- ✅ Pass a minimum test (like boot and basic function on Cuttlefish emulator).
- 🧪 OEMs use them to:
  - Test early driver compatibility.
  - Debug kernel issues.
  - Try out latest upstream features.
- 📥 Can also be fetched from Android's public CI system (internal teams know the URLs, partners get access).

📍 These are for **internal testing only**, not for real user devices.

🧑‍🔧 What OEMs Do with the GKI

Once OEMs download a GKI build:

1. **They add their own kernel modules**:
   - For hardware like fingerprint sensors, cameras, modems, etc.

- These are called **vendor modules** or **DLKMs** (Dynamically Loadable Kernel Modules).

2. **Build their boot.img** using:
   - GKI kernel (boot image).
   - Their ramdisk (init scripts, sepolicy, etc.).
   - Vendor modules packed as `.ko` files.
3. **Run validation tests** (CTS, VTS, etc.).
4. **Push OTA updates** with newer GKI versions (same KMI version or later).

🔄 How to Sync Source and Build GKI (Optional)

If OEMs want to build GKI themselves, Google provides:

```bash
repo init -u https://android.googlesource.com/kernel/manifest
# Add proper XML manifest (e.g. android14-6.1.xml)
mv <manifest_file>.xml .repo/manifests/
repo init -m manifest_file.xml
repo sync
```

This pulls the correct source and tools to reproduce the GKI exactly.

🧯 What if a serious issue is found?

If a critical bug or security flaw is found:

- 🔥 Google may issue an **emergency (out-of-band) GKI release**.
- OEMs must quickly **test** and **ship** this fix through OTA.

Would you like:

- 📈 A flow diagram of this?
- 🧪 Explanation of Cuttlefish or CI setup?
- 🔍 Details about manifest files and branches like `android14-6.1`?

Let me know!

50.

Quarterly certified releases
GKI quarterly releases contain a tested boot.img that includes a Google inserted certificate to attest that the binaries were built from a known source code baseline.

Each quarter, a GKI quarterly release candidate (not certified) is selected after the check-in cut off date, which is usually the second weekly build of that month. After the quarterly release candidate is selected, new changes won't be accepted into that month's release. During the closed window period, only fixes for bugs that cause test failure can be addressed. The release candidate undergoes quality assurance—as described in the GKI qualification section—to ensure compliance tests pass on GSI+GKI build with a reference device as well as cuttlefish.

Emergency respin process
A respin refers to the process of remerging, rebuilding, retesting, and recertifying a binary after a public release of the GKI kernel. You can request a respin of a certified binary for any of the following circumstances:

To update a symbol list.
To apply a fix to a bug, including bugs found during carrier lab approval.
To add a vendor hook.
To apply a patch to an existing feature.
To apply a security patch (after 6 months).
Security patches are automatically merged into a release branch for up to 6 months after the branch's release. After the 6 month cutoff, you must request a respin to apply security patches to a branch.

Respin request guidelines
Before requesting a respin, note the following guidelines:

Respins are allowed only on release branches after an initial public release of a quarterly build has been launched.

Respin requests are accepted only for a given release branch for a maximum of six months after the initial public release. After six months, branches are eligible for respin only for security patches cited in an Android Security Bulletin.

When the LTS requirements , defined by the Android Security Bulletin (ASB) cause the branch to be noncompliant, the branch is deprecated. Respin requests for deprecated branches aren't accepted. The deprecation date for a given GKI release branch is included in the quarterly GKI release build notes under Releases. For future planning, the LTS requirements are updated in May and November annually. For example, the android12-5.10-2023-07 branch (5.10.177) isn't supported for respins after May 1, 2024, because the android12-5.10-2023-07 branch (5.10.177) doesn't comply with the LTS requirements of ASB-2024-05.

Respins are applicable only for urgent bug fixes, symbol list updates, or to apply a patch to fix an existing feature.

All patches going into the quarterly release branch must already be merged into the main GKI development branch. For example, if a patch is required for a respin of android12-5.10-2022-09, it must already be merged into android12-5.10.

You must cherry-pick patches from the main GKI development branch and upload the patch to the quarterly release branch.

In the respin request, you must assign a priority (urgency) to the request. This priority helps the GKI team to better assist partners in a timely manner. For critical or time-sensitive requests, mark priority as P0. For P0 and P1 requests, you must also justify the urgency. The following table provides a mapping of bug priority and time to resolution (ESRT):

Priority ESRT
P0 2 business days
P1 5 business days
P2 10 business days
P3 15 business days
Note: ESRT is defined as the time it takes to deliver a certified GKI binary that contains the approved (by GKI team) and reviewed (by you) fix.
You must submit a separate respin request per release branch. For example, if a respin is needed for both android12-5.10-2022-08 and android12-5.10-2022-09, you must create two respin requests.

After a build is provided and a respin request is marked as fixed, you shouldn't reopen the respin request to add additional CLs. You must submit a new respin request if there are additional patches that need to be merged.

For each CL under consideration, add the following tags.

Bug: the bug ID must be added to the commit message for each CL.
Change-Id: must be identical to the Change-Id of the base branch change.
Note: Progress on the respin request is blocked without these tags.
If a respin request requires your response, and you don't respond within three business days, the priority is downgraded by one level (for example, P0 is downgraded to P1). If you don't respond for two weeks, the bug is marked as Won't Fix (Obsolete).

Here's a **simplified explanation** of the image and text about **GKI Quarterly Certified Releases** and the **Emergency Respin Process**:

✅ GKI Quarterly Release – Made Simple

📅 Timeline Overview (Image)

This diagram shows how GKI releases happen **every month** after a fixed **cut-off date**:

- **KMI Freeze (7/14)**:
  Google freezes the kernel module interface (KMI). No more changes that break compatibility.
- **First GKI Release**:
  The first official kernel binary (boot image) is released after the freeze.
- **Monthly Cycle (Aug–Dec)**:
  - **Check-in Cut Off**: No more feature/driver commits for that month.
  - **GKI Release**: A certified `boot.img` is built, tested, and released.

✅ What is a "Quarterly Certified GKI"?

- It's a **Google-tested boot.img**.
- ✅ Passes all validation tests (CTS, VTS, GSI + Cuttlefish).
- 🔒 Digitally signed by Google.
- 🏭 Can be safely used by OEMs in production devices.

🔁 Emergency Respin – What & When?

A **respin** means **rebuilding and re-releasing** a GKI version *after* it's already public. It's allowed when:

✅ When Can You Request a Respin?

- Critical bug fix (e.g. camera crash, bootloop).
- Security patch (especially after 6 months).
- Adding/modifying a vendor hook.
- Update KMI symbol list.

🚫 When Can't You Request a Respin?

- If the branch is older than 6 months and not security-related.
- If Google marks the branch **deprecated** due to LTS compliance failure.
- If you want to add new features (not allowed in respins).

📋 Rules You Must Follow

1. ✅ Patch must be in the main GKI development branch already.
2. 🧩 Cherry-pick it to the release branch yourself.

3. 🐞 Tag every patch with:
   - `Bug: [bug ID]`
   - `Change-Id: [same as original]`
4. 📒 Assign a priority (P0 = urgent):

Priority	Resolution Time
P0	2 days
P1	5 days
P2	10 days
P3	15 days

5. 🕒 If you don't reply in 3 days, priority is downgraded.
6. 📪 If no response in 2 weeks → marked as **Won't Fix**.

✍️ Example: How OEMs Use This

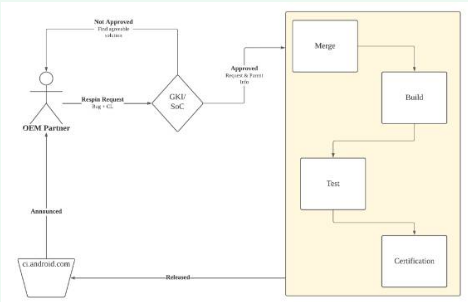
Let's say you ship a phone with:

- `android12-5.10-2022-09` GKI

Then during carrier lab testing you find a modem crash:

- ✅ You submit a **respin request** for that branch.
- 🧩 You cherry-pick your fix.
- 🧪 Google tests it and sends you a new signed boot.img.

Let me know if you want this as a **flow diagram or checklist**, or if you'd like help writing an actual **respin request format**.

51.


Submit a respin request
The following diagram shows the respin process. The process begins when the OEM Partner (you) submits the respin request.

To enter into the respin process:

Fill out the GKI Respin request form. and reach out to your Google Technical Account Manager immediately. This form creates a GKI respin request bug. Respin request bugs are visible to you (the requester), the GKI team, and specific individuals that you add to the bug's CC list.
If you already have a fix, the request should point to the patch submittal in AOSP so Google can review it. If submitting the patch isn't feasible, the patch must be attached as a text file to the request.
If you don't have a fix, the request must contain as much information as possible, including kernel version number and logs, so Google can help debug the issue.
The Google GKI team reviews the request and approves it or assigns it back to you if more information is needed.
After a fix is agreed upon, the Google GKI team code reviews (CR+2) the change. The review begins the ESRT timeframe. The GKI team merges, builds, tests for regression, and certifies the change.
The binary is released to ci.android.com. The ESRT timeframe ends and the Google GKI team marks the request as fixed and reference the respin build. The respin build is also be posted on the Generic Kernel Image (GKI) release builds page.
GKI qualifications
Types of GKI builds	Quality enforcement	Notes
Weekly	Cuttlefish testing
Boot
Subset of VTS
Subset of CTS
Not certified. Only for testing and
device bring up.
Can't be used for launching devices.
Quarterly (certified)	Cuttlefish testing
Boot
VTS
CTS
Reference hardware testing
Boot
VTS
CTS
Respins (certified)	Cuttlefish testing
Boot
VTS
Subset of CTS
Reference device testing
Boot
VTS
Built on top of a GKI certified build.
The build is certified after qualification.
Where to obtain build artifacts
Artifacts for all the releases can be obtained from ci.android.com.

You can find more information on the CI, including the test results on the Android Continuous Integration dashboard.

FAQs
Here are some frequently asked questions related to the GKI release process.

Is it possible to build a new GKI binary based on an already released GKI?

Yes, this is known as a respin. The respin process is supported as long as the released GKI build (on which the respin is requested) is compliant with LTS requirements in the Android Security Bulletin (ASB).

Is it possible to reproduce GKI binaries?
Yes, here's an example:

GKI 2.0
5.10 kernel prebuilts from build 7364300
https://ci.android.com/builds/submitted/7364300/kernel_aarch64/latest
To reproduce the example, download manifest_$id.xml and execute the following command:

repo init -u https://android.googlesource.com/kernel/manifest
mv manifest_7364300.xml .repo/manifests
repo init -m manifest_7364300.xml --depth=1
repo sync
# build the GKI images
# You may want to use LTO=thin to build faster for development
BUILD_CONFIG=common/build.config.gki.aarch64 build/build.sh
# (optional) build virtual platform modules
BUILD_CONFIG=common-modules/virtual-device/build.config.virtual_device.aarch64 build/build.sh
You can retrieve your GKI artifact copy from out/.../dist.

Has the GKI binary (including the emergency spin patch) been built on the latest codebase?
No. Respins only contain patches that are on top of the quarterly certified kernels that have been chosen. These respins contain all launch blocking bug fixes reported until any given time by OEMs using the corresponding base quarterly release. See the following example of how this type of scenario happens.

OEM1 and OEM2 decide to use the GKI binary release from November 2021.
OEM1 and OEM2 find issues that require patches for support. These patches may be different or may be the same.
The respins over top of the November 2021 binary have launch blocking fixes reported by both OEM1 and OEM2 during the respin window, but nothing more.
The issues mentioned in the second bullet are also included in subsequent GKI quarterly releases.
The October respin has all OEM submitted patches, but other OEM patches affect us, because they haven't been specifically tested with our products. Is it possible to only include our patch?
This isn't possible. A "per-OEM" respin path isn't scalable. Instead, the GKI team scrutinizes every single change that goes into respin builds, and tests the changes with all available hardware before creating a new build. If the GKI team finds that the issue is specific to an OEM, device, or model, the GKI team can ensure that the code added by the change only executes on the device, model, or SKU being affected.

The major benefit from unified respins is that every device that's using the same release base benefits from one another, especially if the bugs they discover are generic and applicable to all users. Core kernel bugs found in carrier testing is a specific example of this concept.

Are there situations where Google provides specific information about OEM patches and issue scenarios, so that OEMs can evaluate the impact and risk of implementing the patches with their products?
Google won't ever add a change to a respin build until the problem is understood and all of the details have been collected. This is seen in the changelog (commit message). Google does not reveal what specific device it affects, but OEMs can always find the issue description and solution in the changelog.

Here's a **simple explanation** of the GKI **respin request process**, based on the diagram and text:

🔁 What is a GKI Respin?

A **respin** means building a new GKI image (boot.img) on top of a certified release, usually to fix **urgent bugs**, apply **security patches**, or **update vendor hooks**.

🧭 Simple Flow (Diagram Explained)

🔹 Step-by-Step

1. **OEM Partner Submits Request**
    - You (the OEM/SoC vendor) find a critical issue.

- You prepare a **bug report** + **CL (patch)**.
- You submit a **Respin Request** to Google (GKI/SoC team).

2. **Google GKI Team Review**
   - Google **reviews the request**.
   - If **approved**, they move forward.
   - If **not approved**, they ask you to fix or clarify.
3. **Build Process Begins**
   - Patch is merged into the **respin branch**.
   - The GKI team runs the full pipeline:
     - **Merge**
     - **Build**
     - **Test** (CTS, VTS, Cuttlefish, etc.)
     - **Certification**
4. **Release**
   - The new boot.img is **uploaded to** ci.android.com.
   - OEM is **notified (Announced)** and can download and use it.

📋 Summary of Respin Request Rules

Step	Requirement
🔧 Patch Required	Must already be in the main GKI branch (or attach .txt if not)
🧾 Bug Required	Must have a bug report with logs, kernel version, etc.
🔁 Validity	Only valid within 6 months of release
🚫 Not allowed for	Feature additions, deprecated branches
🎯 Patch Tagging	Must include `Bug:` and `Change-Id:` in commit
🕒 Response Window	No reply in 3 days → downgraded, 2 weeks → closed

---

🧪 GKI Build Quality Levels

Build Type	Testing	Certified?	Use Case
Weekly	Cuttlefish + basic CTS/VTS	❌ No	For testing/dev only
Quarterly	Cuttlefish + full CTS/VTS + HW	✅ Yes	Launch ready
Respin	Cuttlefish + subset of CTS + HW	✅ Yes	Urgent fixes on top of certified builds

---

🔍 Example Use Case

- You use `android13-5.10-2023-09` GKI build.
- You find a modem crash during carrier testing.
- You submit a respin request.
- Google reviews, tests, and posts a **new boot.img** with your fix.

🛑 OEM-specific respins?

No. Google does **not allow per-OEM respins**. All patches in a respin are validated for **general impact** and made available to all using that release. This improves:

- 🛡️ Stability
- 🧪 Cross-device bug fixes
- 📉 Fragmentation

Let me know if you want a visual cheat sheet, a checklist to submit your own respin, or help writing a real patch/CL.

52. Maintain a stable kernel module interface

bookmark_border
It's critical to maintain a stable kernel module interface (KMI) for vendor modules. The GKI kernel is built and shipped in binary form and vendor-loadable modules are built in a separate tree. The resulting GKI kernel and vendor modules must work as though they were built together.

Generally, the Linux community has frowned on the notion of in-kernel ABI stability for the mainline kernel. In the face of different toolchains, configurations, and an ever-evolving Linux mainline kernel, it isn't feasible to maintain a stable KMI in mainline. However, it's possible to maintain a stable KMI in the highly-constrained GKI environment with these constraints:

Only a single configuration, gki_defconfig, can be used to build the kernel.

The KMI is only stable within the same LTS and Android version of a kernel, such as android14-6.1, android15-6.6 or android16-6.12.

No KMI stability is maintained for android-mainline.
Only the specific Clang toolchain supplied in AOSP and defined for the corresponding branch is used for building kernel and modules.

Only symbols known to be used by modules as specified in a symbol list are monitored for stability and considered KMI symbols.

The corollary is that vendor modules must use only KMI symbols. This constraint is enforced by failing module loads if non-KMI-symbols are required.
After the KMI branch is frozen, changes are allowed but can't break the KMI. These changes include the following:

Config changes
Kernel code changes
Toolchain changes (including updates)
Use the hermetic build process and LLVM toolchain
The hermetic build process ensures a stable KMI by having repo manifests in kernel/manifest completely describe the build environment. For example, the manifest for android16-6.12 includes the toolchain, build system, and everything else required to build the Generic Kernel Image (GKI) kernel. The build configuration, primarily BUILD.bazel, ensures that the included tools are used correctly to generate consistent build results.

Using a hermetic build process also ensures that the ABI description for the tree is consistent whether generated by Google (for example, gki/aarch64/abi.stg for android16-6.12) or generated in a local tree that includes the vendor modules. The tools to create and compare the ABI description for the Kernel Module Interface (KMI) are also provided as part of the repo described by the manifest.

The toolchain used to build the GKI kernel must be completely compatible with the toolchain used to build vendor modules. As of Android 10, all Android kernels must be built with an LLVM toolchain. With GKI, the LLVM toolchain used to build product kernels and vendor modules must generate the same ABI as the LLVM toolchain from AOSP and partners must ensure that the KMI is compatible with the GKI kernel. Using the provided build tools is strongly encouraged as they provide the best compatibility.

What's next?
For instructions on building the kernel using the hermetic build process and LLVM toolchain, refer to refer to Build kernels.

For instructions on how to monitor the ABI and fix issues, refer to Android Kernel ABI Monitoring

Here's a **simple explanation** of how Android **maintains a stable Kernel Module Interface (KMI)** in GKI (Generic Kernel Image):

🧩 What Is KMI?

The **Kernel Module Interface (KMI)** is like an *agreed handshake* between:

- The **GKI kernel** (prebuilt binary from Google)
- The **vendor modules** (your device-specific drivers compiled separately)

They **must work together** — even if they were built at different times, in different places.

⚠️ Why Is It Hard in Linux?

In mainline Linux:

- Kernels change fast (new APIs, toolchains, configs)
- Kernel interfaces (symbols, data structures) keep evolving
- So, maintaining a **stable ABI (Application Binary Interface)** is not guaranteed

But in Android GKI:

> ✅ A *controlled environment* makes a stable KMI possible.

✅ How Android Keeps KMI Stable

Constraint	Why It Matters
🧱 Use **`gki_defconfig` only**	All builds use the same kernel configuration
📦 One **Clang toolchain** per kernel branch	Ensures ABI is always generated in the same way
🔍 Only **monitored KMI symbols** can be used	Prevents accidental breakage — non-KMI symbols are blocked
❄️ After KMI Freeze → No breaking changes	Kernel can evolve but must not break the interface
🔐 Use **hermetic build system**	Ensures the full build environment is consistent and reproducible

---

🧪 What Happens If You Break KMI?

If a vendor module tries to use a symbol **not in the KMI list**, it **fails to load**.
This protects device stability.

🔧 What Is the Hermetic Build Process?

A **hermetic build** means:

- Build is **isolated** and **reproducible**
- Every tool (Clang, kernel config, build scripts) is **locked and tracked**
- **`repo init`** with manifest from **`kernel/manifest`** sets up exactly the right versions

So:

> You and Google build the same kernel, same way, same ABI → No surprises.

🛠️ Tools That Help

- 📄 ABI symbol list (**`abi.stg`**) defines which kernel symbols are safe for vendors
- 🔧 ABI diff tools check if a change broke compatibility
- 🧰 These tools are part of the GKI **`repo`** (you get them when you sync)

🚧 After KMI Freeze...

Once a KMI is **frozen** (like **`android16-6.12`**), you can still:

- Change config
- Update toolchain
- Fix kernel bugs

...as long as **you don't break the KMI**.

🔗 What to Read Next

- How to **build with hermetic toolchain**
- How to **monitor kernel ABI**

Let me know if you want a **visual diagram** of this, or a **step-by-step guide** to setting up the build with KMI protection.

53. why ABI matter here? explain in simple

Great question. Let's break it down **simply**.

🔄 What is ABI?

ABI stands for **Application Binary Interface**.

It's like a **contract** between two compiled components.
In the GKI world, that means:

📦 **GKI Kernel (precompiled binary from Google)**
🤝
🔌 **Vendor modules (camera driver, GPU driver, etc.)**

The ABI defines **how**:

- Functions are called
- Data structures look
- Memory layout is handled

So both sides can **talk correctly**, **even when built separately**.

📌 Why ABI Matters in GKI?

In Android GKI:

🔧 GKI kernel is built by Google and shared in binary form

→ You **don't recompile** or modify it.

🧩 Vendor modules (your drivers) are built by you

→ You compile them **later**, maybe with a different setup.

If the **ABI changes** and your module expects the *old* layout:

- Function call mismatches
- Data structure size is wrong
- ❌ **Crash at boot** or weird bugs

🎯 So ABI Stability = Safe Updates

ABI ensures:

- ✅ GKI updates don't break your vendor modules
- ✅ You can update the kernel separately from vendor code
- ✅ Devices are more maintainable and secure
- ✅ Modules don't need to be rebuilt every time

📦 Example

Let's say kernel exports this symbol:

```c
struct CameraConfig {
    int exposure;
```

```
    int focus;
};
```

You build your module using that.

Later, Google changes it:

```c
struct CameraConfig {
    int exposure;
    int focus;
    int iso;  // ← new field added
};
```

Now your old module doesn't know about **`iso`**. Memory layouts **don’t match**, and it could cause:

- 💥 Crashes
- 🔒 Security bugs
- 🛑 Driver won't load

This is why Android locks the ABI with a **symbol list**.

✅ Final Summary

Term	Meaning
ABI	Rules for how binaries (kernel & modules) work together
Why it matters	To prevent crashes when kernel updates
How GKI handles it	Freezes a symbol list and strictly checks for ABI stability
Vendor rule	Use only approved symbols (called **KMI symbols**)

Let me know if you want an image for this — it’s a common interview topic too.

54. ABI vs KMI

Great question! Let's simplify the difference between **ABI** and **KMI**, especially in the **Android GKI** context.

🧩 What is ABI?

ABI (Application Binary Interface)
➡️ General term that defines **how binaries interact at runtime**, including:

- How functions are called (calling convention)
- How data structures are laid out in memory
- How return values are passed
- How system calls are made

ABI applies to:

- User apps ↔ Kernel
- Kernel ↔ Kernel modules
- Any compiled binary ↔ Another binary

So, **ABI** is a broad concept.

🧱 What is KMI?

KMI (Kernel Module Interface)
➡️ A **specific subset of ABI** in the kernel:
It's the **set of symbols** (functions, variables) that **kernel modules** can use when loaded dynamically.

KMI is:

- Kernel ↔ Kernel Modules (like vendor drivers)
- Defined by a **symbol list** (i.e. `abi_gki_aarch64.xml`)
- Monitored & frozen in GKI after a certain release

Think of **KMI as the ABI that kernel modules depend on.**

🔍 Main Differences

Feature	ABI	KMI
Scope	Broad: user space, kernel, modules	Narrow: kernel <-> vendor kernel modules
Use Case	All binary interactions	Only GKI kernel ↔ vendor modules
Example	libc calls, syscalls, driver functions	`usb_register_driver()`, `mmc_detect_change()`
Change Allowed?	Sometimes yes	Never after freeze! (in GKI)
Stability Enforcement	Optional or best effort	Strictly enforced in Android GKI
Maintained By	OS / compiler / platform	Android GKI team (with symbol list)

🧠 In Simple Words

> **ABI** is the general rules of how any compiled programs talk.
> **KMI** is the special contract between **Android's GKI kernel** and **vendor drivers**.

🏁 Why KMI is Critical in Android GKI?

Because:

- GKI kernel is built by Google
- Vendor modules are built separately by OEMs
- They must work **without recompiling everything**
- So KMI (symbol list) must be **stable**

Let me know if you'd like a **diagram** showing the difference!

55. Android kernel ABI monitoring

bookmark_border
You can use application binary interface (ABI) monitoring tooling, available in Android 11 and higher, to stabilize the in-kernel ABI of Android kernels. The tooling collects and compares ABI representations from existing kernel binaries (vmlinux+ GKI modules). These ABI representations are the .stg files and the symbol lists. The interface on which the representation gives a view is called the Kernel Module Interface (KMI). You can use the tooling to track and mitigate changes to the KMI.

The ABI monitoring tooling is developed in AOSP and uses STG (or libabigail in Android 13 and lower) to generate and compare representations.

This page describes the tooling, the process of collecting and analyzing ABI representations, and the usage of such representations to provide stability to the in-kernel ABI. This page also provides information for contributing changes to the Android kernels.

Process
Analyzing the kernel's ABI takes multiple steps, most of which can be automated:

Build the kernel and its ABI representation.
Analyze ABI differences between the build and a reference.
Update the ABI representation (if required).
Work with symbol lists.
The following instructions work for any kernel that you can build using a supported toolchain (such as the prebuilt Clang toolchain). repo manifests are available for all Android common kernel branches and for several device-specific kernels, they verify that the correct toolchain is used when you build a kernel distribution for analysis.

Symbol lists
The KMI doesn't include all symbols in the kernel or even all of the 30,000+ exported symbols. Instead, the symbols that can be used by vendor modules are explicitly listed in a set of symbol list files maintained publicly in the kernel tree (gki/{ARCH}/symbols/* or android/abi_gki_{ARCH}_* in Android 15 and lower). The union of all the symbols in all of the symbol list files defines the set of KMI symbols maintained as stable. An example symbol list file is gki/aarch64/symbols/db845c, which declares the symbols required for the DragonBoard 845c.

Only the symbols listed in a symbol list and their related structures and definitions are considered part of the KMI. You can post changes to your symbol lists if the symbols you need aren't present. After new interfaces are in a symbol list, and are part of the KMI description, they're maintained as stable and must not be removed from the symbol list or modified after the branch is frozen.

Each Android Common Kernel (ACK) KMI kernel branch has its own set of symbol lists. No attempt is made to provide ABI stability between different KMI kernel branches. For example, the KMI for android12-5.10 is completely independent of the KMI for android13-5.10.

ABI tools use KMI symbol lists to limit which interfaces must be monitored for stability. Vendors are expected to submit and update their own symbol lists to verify that the interfaces they rely on maintain ABI compatibility. For example, to see a list of symbol lists for the android16-6.12 kernel, refer to https://android.googlesource.com/kernel/common/+/refs/heads/android16-6.12/gki/aarch64/symbols

A symbol list contains the symbols reported to be needed for the particular vendor or device. The complete list used by the tools is the union of all of the KMI symbol list files. ABI tools determine the details of each symbol, including function signature and nested data structures.

When the KMI is frozen, no changes are allowed to the existing KMI interfaces; they're stable. However, vendors are free to add symbols to the KMI at any time as long as additions don't affect the stability of the existing ABI. Newly added symbols are maintained as stable as soon as they're cited by a KMI symbol list. Symbols shouldn't be removed from a list for a kernel unless it can be confirmed that no device has ever shipped with a dependency on that symbol.

You can generate a KMI symbol list for a device using the instructions from How to work with symbol lists. Many partners submit one symbol list per ACK, but this isn't a hard requirement. If it helps with maintenance, you can submit multiple symbol lists.

Extend the KMI
While KMI symbols and related structures are maintained as stable (meaning changes that break stable interfaces in a kernel with a frozen KMI cannot be accepted) the GKI kernel remains open to extensions so that devices shipping later in the year don't need to define all their dependencies before the KMI is frozen. To extend the KMI, you can add new symbols to the KMI for new or existing exported kernel functions, even if the KMI is frozen. New kernel patches might also be accepted if they don't break the KMI.

About KMI breakages
A kernel has sources and binaries are built from those sources. ABI-monitored kernel branches include an ABI representation of the current GKI ABI (in the form of a .stg file). After the binaries (vmlinux, Image and any GKI modules) are built, an ABI representation can be extracted from the binaries. Any change made to a kernel source file can affect the binaries and in turn also affect the extracted .stg. The AbiAnalyzer analyzer compares the committed .stg file with the one extracted from build artefacts and sets a Lint-1 label on the change in Gerrit if it finds a semantic

difference.

Handle ABI breakages
As an example, the following patch introduces a very obvious ABI breakage:

```
diff --git a/include/linux/mm_types.h b/include/linux/mm_types.h
index 42786e6364ef..e15f1d0f137b 100644
--- a/include/linux/mm_types.h
+++ b/include/linux/mm_types.h
@@ -657,6 +657,7 @@ struct mm_struct {
                ANDROID_KABI_RESERVE(1);
        } __randomize_layout;

+       int tickle_count;
        /*
         * The mm_cpumask needs to be at the end of mm_struct, because it
         * is dynamically sized based on nr_cpu_ids.
```

When you run build ABI with this patch applied, the tooling exits with a non-zero error code and reports an ABI difference similar to this:

```
function symbol 'struct block_device* I_BDEV(struct inode*)' changed
  CRC changed from 0x8d400dbd to 0xabfc92ad

function symbol 'void* PDE_DATA(const struct inode*)' changed
  CRC changed from 0xc3c38b5c to 0x7ad96c0d

function symbol 'void __ClearPageMovable(struct page*)' changed
  CRC changed from 0xf489e5e8 to 0x92bd005e

... 4492 omitted; 4495 symbols have only CRC changes

type 'struct mm_struct' changed
  byte size changed from 992 to 1000
  member 'int tickle_count' was added
  member 'unsigned long cpu_bitmap[0]' changed
    offset changed by 64
```

ABI differences detected at build time
The most common reason for errors is when a driver uses a new symbol from the kernel that isn't in any of the symbol lists.

If the symbol isn't included in your symbol list, then you need to first verify that it's exported with EXPORT_SYMBOL_GPL(symbol_name) and then update the symbol list and ABI representation. For example, the following changes add the new Incremental FS feature to the android-12-5.10 branch, which includes updating the symbol list and ABI representation.

Feature change example is in aosp/1345659.
Symbol list example is in aosp/1346742.
ABI representation change example is in aosp/1349377.
If the symbol is exported (either by you or it was previously exported) but no other driver is using it, you might get a build error similar to the following.

```
Comparing the KMI and the symbol lists:
+ build/abi/compare_to_symbol_list out/$BRANCH/common/Module.symvers out/$BRANCH/common/abi_symbollist.raw
ERROR: Differences between ksymtab and symbol list detected!
Symbols missing from ksymtab:
Symbols missing from symbol list:
 - simple_strtoull
```

To resolve, update the KMI symbol list in both your kernel and the ACK (see Update the ABI representation). For an example of updating a symbol list and the ABI representation in the ACK, refer to aosp/1367601.

Here's a **simple explanation** of **Android Kernel ABI Monitoring** and why it's important:

🧠 What is Android Kernel ABI Monitoring?

It's a **system to ensure that changes in the Android kernel won't break vendor drivers** (like your Wi-Fi, camera, etc.) that are built separately and depend on certain kernel interfaces.

🔍 Why It's Needed

In Android GKI (Generic Kernel Image):

- Google builds the kernel (boot.img).
- OEMs build their own vendor modules (drivers).
- These two must work together — **even if built separately.**

This is only possible if **interfaces used by vendor modules stay stable** — that's where **ABI monitoring** comes in.

🧱 What Does ABI Monitoring Do?

It checks for:

- Function signature changes
- Struct layout changes
- Symbol list updates
- Any binary-level change in the **Kernel Module Interface (KMI)**

👉 If a change is detected, it **fails the build** or **flags the patch**.

⚙️ Tools Used

- `.stg` files = ABI snapshot of kernel binary (like a fingerprint)
- **AbiAnalyzer** = compares your build ABI with the stored `.stg`
- **Symbol lists** = list of kernel functions (`EXPORT_SYMBOL`) allowed to be used by vendor drivers

🔁 Typical ABI Monitoring Workflow

1. ✅ Build the kernel → generates `.stg` and `Module.symvers`.
2. 🔍 ABI Tool checks `.stg` against committed `.stg` in the repo.
3. ❌ If any changes found → error or Gerrit Lint warning (`Lint-1`).
4. 🛠️ If symbol missing → update your symbol list (`gki/aarch64/symbols/your-device`).
5. ✅ Submit patch if ABI change is intentional and doesn't break stability.

⚠️ What Breaks ABI?

Example (struct change):

```diff
 struct mm_struct {
     ...
+    int tickle_count;  // 🔴 added field
 };
```

This changes struct size → ABI breakage. ABI tool detects and reports:

```vbnet
type 'struct mm_struct' changed
  byte size changed from 992 to 1000
```

🧪 Types of ABI Errors

- Function CRC (hash) changed
- Struct size or field offset changed
- Symbol used but not in symbol list
- Symbol exported, but not used or listed

🧩 KMI & Symbol List Explained

- **KMI** = The official, monitored part of the kernel ABI for vendor drivers
- **Symbol list** = File listing all allowed symbols (functions/structs) vendor drivers can use
- Tool only monitors these listed symbols

Example symbol list path:

```bash
kernel/common/gki/aarch64/symbols/db845c
```

✅ Summary in Simple Terms

Term	Meaning
ABI Monitoring	Ensures kernel changes don't break binary compatibility
.stg	Binary fingerprint of kernel symbols & types
Symbol List	List of allowed symbols for vendor drivers
AbiAnalyzer	Tool that compares ABI vs old ABI
KMI	Kernel Module Interface = stable ABI surface for modules
ABI Breakage	Happens if symbol/struct/function changes unexpectedly

Let me know if you want a **visual diagram** showing this monitoring pipeline!

56. Resolve kernel ABI breakages
You can handle kernel ABI breakages by refactoring the code to not change the ABI or updating the ABI representation. Use the following chart to determine the best approach for your situation.

Refactor code to avoid ABI changes
Make every effort to avoid modifying the existing ABI. In many cases, you can refactor your code to remove changes that affect the ABI.

Refactoring struct field changes. If a change modifies the ABI for a debug feature, add an #ifdef around the fields (in the structs and source references) and make sure the CONFIG used for the #ifdef is disabled for the production defconfig and gki_defconfig. For an example of how a debug config can be added to a struct without breaking the ABI, refer to this patchset.

Refactoring features to not change the core kernel. If new features need to be added to ACK to support the partner modules, try to refactor the ABI part of the change to avoid modifying the kernel ABI. For an example of using the existing kernel ABI to add additional capabilities without changing the kernel ABI refer to aosp/1312213.

Fix a broken ABI on Android Gerrit
If you didn't intentionally break the kernel ABI, then you need to investigate, using the guidance provided by the ABI monitoring tooling. The most common causes of breakages are changed data structures and the associated symbol CRC changes, or due to config option changes that lead to any of the aforementioned. Begin by addressing the issues found by the tool.

You can reproduce the ABI findings locally, see Build the kernel and its ABI representation.

About Lint-1 labels
If you upload changes to a branch containing a frozen or finalized KMI, the changes must pass the AbiAnalyzer to verify that they don't affect the stable ABI in an incompatible way. During this process, the AbiAnalyzer looks for the ABI report that's created during the build (an extended build that performs the normal build and then some ABI extraction and comparison steps.

If the AbiAnalyzer finds a non-empty report it sets the Lint-1 label and the change is blocked from submittal until resolved; until the patchset receives a Lint+1 label.

Note: Patchsets can also receive a Lint-1 label by failing other analyzer checks.
Note: The analyzer also runs on the newer branches, but the results aren't valid until the KMI on a branch is finalized.
Update the kernel ABI
If modifying the ABI is unavoidable, then you must apply your code changes, the ABI representation, and symbol list to the ACK. To get Lint to remove the -1 and not break GKI compatibility, follow these steps:

Upload code changes to the ACK.

Wait to receive a Code-Review +2 for the patchset.

Update the reference ABI representation.

Merge your code changes and the ABI update change.

Note: If you haven't downloaded the ACK, refer to building Android kernels.
Upload ABI code changes to the ACK
Updating the ACK ABI depends on the type of change being made.

If an ABI change is related to a feature that affects CTS or VTS tests, the change can usually be cherry-picked to ACK as is. For example:

aosp/1289677 is needed for audio to work.
aosp/1295945 is needed for USB to work.
If an ABI change is for a feature that can be shared with the ACK, that change can be cherry-picked to ACK as is. For example, the following changes aren't needed for CTS or VTS test but are OK to be shared with ACK:

aosp/1250412 is a thermal feature change.
aosp/1288857 is an EXPORT_SYMBOL_GPL change.
If an ABI change introduces a new feature that doesn't need to be included in the ACK, you can introduce the symbols to ACK using a stub as as described in the following section.

Use stubs for ACK
Stubs must be necessary only for core kernel changes that don't benefit the ACK, such as performance and power changes. The following list

details examples of stubs and partial cherry-picks in ACK for GKI.

Core-isolate feature stub (aosp/1284493). The capabilities in ACK isn't necessary, but the symbols need to be present in ACK for your modules to use these symbols.

Placeholder symbol for vendor module (aosp/1288860).

ABI-only cherry-pick of per-process mm event tracking feature (aosp/1288454). The original patch was cherry-picked to ACK and then trimmed to include only the necessary changes to resolve the ABI diff for task_struct and mm_event_count. This patch also updates the mm_event_type enum to contain the final members.

Partial cherry-pick of thermal struct ABI changes that required more than just adding the new ABI fields.

Patch aosp/1255544 resolved ABI differences between the partner kernel and ACK.

Patch aosp/1291018 fixed the functional issues found during GKI testing of the previous patch. The fix included initializing the sensor parameter struct to register multiple thermal zones to a single sensor.

CONFIG_NL80211_TESTMODE ABI changes (aosp/1344321). This patch added the necessary struct changes for ABI and made sure the additional fields didn't cause functional differences, enabling partners to include CONFIG_NL80211_TESTMODE in their production kernels and still maintain GKI compliance.

Caution: Partial cherry-picks of features can be tricky and must be done only for proprietary or hardware-specific features (or features that might break other partner modules in the ACK).
Enforce the KMI at runtime
The GKI kernels use the TRIM_UNUSED_KSYMS=y and UNUSED_KSYMS_WHITELIST=<union of all symbol lists> configuration options, which limit the exported symbols (such as symbols exported using EXPORT_SYMBOL_GPL()) to those listed on a symbol list. All other symbols are unexported, and loading a module requiring an unexported symbol is denied. This restriction is enforced at build time and missing entries are flagged.

For development purposes, you can use a GKI kernel build that doesn't include symbol trimming (meaning all usually exported symbols can be used). To locate these builds, look for the kernel_debug_aarch64 builds on ci.android.com.

Enforce the KMI using module versioning
The Generic Kernel Image (GKI) kernels use module versioning (CONFIG_MODVERSIONS) as an additional measure to enforce KMI compliance at runtime. Module versioning can cause cyclic redundancy check (CRC) mismatch failures at module load time if the expected KMI of a module doesn't match the vmlinux KMI. For example, the following is a typical failure that occurs at module load time due to a CRC mismatch for the symbol module_layout():

init: Loading module /lib/modules/kernel/.../XXX.ko with args ""
XXX: disagrees about version of symbol module_layout
init: Failed to insmod '/lib/modules/kernel/.../XXX.ko' with args ''
Uses of module versioning
Module versioning is useful for the following reasons:

Module versioning catches changes in data structure visibility. If modules change opaque data structures, that is, data structures that aren't part of the KMI, they break after future changes to the structure.

As an example, consider the fwnode field in struct device. This field MUST be opaque to modules so that they can't make changes to fields of device->fw_node or make assumptions about its size.

However, if a module includes <linux/fwnode.h> (directly or indirectly), then the fwnode field in the struct device is no longer opaque to it. The module can then make changes to device->fwnode->dev or device->fwnode->ops. This scenario is problematic for several reasons, stated as follows:

It can break assumptions the core kernel code is making about its internal data structures.

If a future kernel update changes the struct fwnode_handle (the data type of fwnode), then the module no longer works with the new kernel. Moreover, stgdiff won't show any differences because the module is breaking the KMI by directly manipulating internal data structures in ways that can't be captured by only inspecting the binary representation.

A current module is deemed KMI-incompatible when it is loaded at a later date by a new kernel that's incompatible. Module versioning adds a run-time check to avoid accidentally loading a module that isn't KMI-compatible with the kernel. This check prevents hard-to-debug runtime issues and kernel crashes that might result from an undetected incompatibility in the KMI.

Enabling module versioning prevents all these issues.

Check for CRC mismatches without booting the device
stgdiff compares and reports CRC mismatches between kernels along with other ABI differences.

In addition, a full kernel build with CONFIG_MODVERSIONS enabled generates a Module.symvers file as part of the normal build process. This file has one line for every symbol exported by the kernel (vmlinux) and the modules. Each line consists of the CRC value, symbol name, symbol namespace, the vmlinux or module name that's exporting the symbol, and the export type (for example, EXPORT_SYMBOL versus EXPORT_SYMBOL_GPL).

You can compare the Module.symvers files between the GKI build and your build to check for any CRC differences in the symbols exported by vmlinux. If there is a CRC value difference in any symbol exported by vmlinux and that symbol is used by one of the modules you load in your device, the module doesn't load.

If you don't have all the build artifacts, but do have the vmlinux files of the GKI kernel and your kernel, you can compare the CRC values for a specific symbol by running the following command on both the kernels and comparing the output:

```
nm <path to vmlinux>/vmlinux | grep __crc_<symbol name>
```

For example, the following command checks the CRC value for the module_layout symbol:

```
nm vmlinux | grep __crc_module_layout
0000000008663742 A __crc_module_layout
```

Resolve CRC mismatches
Note: The CRC calculation and its inputs changed significantly between Android 15 (genksyms) and Android 16 (gendwarfksyms). While the investigative steps required are similar, the actual resolution of CRC mismatches is very different.
Use the following steps to resolve a CRC mismatch when loading a module:

Build the GKI kernel and your device kernel using the --kbuild_symtypes option as shown in the following command:

```
tools/bazel run --kbuild_symtypes //common:kernel_aarch64_dist
```

This command generates a .symtypes file for each .o file. See KBUILD_SYMTYPES in Kleaf for details.

For Android 13 and lower build the GKI kernel and your device kernel by prepending KBUILD_SYMTYPES=1 to the command you use to build the kernel, as shown in the following command:

```
KBUILD_SYMTYPES=1 BUILD_CONFIG=common/build.config.gki.aarch64 build/build.sh
```

When using build_abi.sh, the KBUILD_SYMTYPES=1 flag is implicitly set already.

Find the .c file in which the symbol with CRC mismatch is exported, using the following command:

```
git -C common grep EXPORT_SYMBOL.*module_layout
kernel/module/version.c:EXPORT_SYMBOL(module_layout);
```

The .c file has a corresponding .symtypes file in the GKI, and your device kernel build artifacts. Locate the .symtypes file using the following commands:

```
cd bazel-bin/common/kernel_aarch64/symtypes
ls -1 kernel/module/version.symtypes
```

In Android 13 and lower, using the legacy build scripts, the location is likely to be either out/$BRANCH/common or out_abi/$BRANCH/common.

Each .symtypes file is a plain text file consisting of type and symbol descriptions:

Each line is of the form key description where the description can refer to other keys in the same file.

Keys like [s|u|e|t]#foo refer to [struct|union|enum|typedef] foo. For example:

```
t#bool typedef _Bool bool
```

Keys with no x# prefix are just symbol names. For example:

```
find_module s#module * find_module ( const char * )
```

Compare the two files and fix all the differences.

It is best to generate symtypes with a build just before the problematic change and then at the problematic change. Saving all the files means they can be compared in bulk.

For example,

```
for f in $(find good bad -name '*.symtypes' | sed -r 's;^(good|bad)/;;' | LANG=C sort -u); do
  diff -N -U0 --label good/"$f" --label bad/"$f" <(LANG=C sort good/"$f") <(LANG=C sort bad/"$f")
done
```

Otherise, just compare the specific files of interest.

Note: Any keys which are added (or removed) can be ignore for the purposes of investigation. We only care about keys whose descriptions have changed.
Note: In Android 16 and higher, the order of the keys isn't deterministic, so the files should be sorted before comparison. There are also minor differences in the description syntax.
Note: Preferably, enable line wrapping in your diff tool. For example, use :set wrap when using vimdff.
Case 1: Differences due to data type visibility
A new #include can pull a new type definition (say of struct foo) into a source file, In these cases, its description in the corresponding .symtypes file will change from an empty structure_type foo { } to a full definition.

Note: In Android 15 and lower, the old description was struct foo { UNKNOWN }.
This will affect all the CRCs of all symbols in the .symtypes file whose descriptions depend directly or indirectly on the type definition.

For example, adding the following line to the include/linux/device.h file in your kernel causes CRC mismatches, one of which is for module_layout():

```
 #include <linux/fwnode.h>
```

Comparing the module/version.symtypes for that symbol, exposes the following differences:

```
 $ diff -u <GKI>/kernel/module/version.symtypes <your kernel>/kernel/module/version.symtypes
  --- <GKI>/kernel/module/version.symtypes
  +++ <your kernel>/kernel/module/version.symtypes
  @@ -334,12 +334,15 @@
  ...
  -s#fwnode_handle structure_type fwnode_handle { }
  +s#fwnode_reference_args structure_type fwnode_reference_args { s#fwnode_handle * fwnode ; unsigned int nargs ; t#u64 args [ 8 ] ; }
  ...
```

If the GKI kernel has the full type definition, but your kernel is missing it (very unlikely), then merge the latest Android Common Kernel into your kernel so that you are using the latest GKI kernel base.

In most cases, the GKI kernel is missing the full type definition in .symtypes, but your kernel has it due to additional #include directives.

Resolution for Android 16 and higher
Make sure that the affected source file includes the Android KABI stabilisation header:

```
#include <linux/android_kabi.h>
```

For each affected type, add ANDROID_KABI_DECLONLY(name); at global scope to the affected source file.

For example, if the symtypes diff was this:

```
--- good/drivers/android/vendor_hooks.symtypes
+++ bad/drivers/android/vendor_hooks.symtypes
@@ -1051 +1051,2 @@
-s#ubuf_info structure_type ubuf_info { }
+s#ubuf_info structure_type ubuf_info { member pointer_type { const_type { s#ubuf_info_ops } } ops data_member_location(0) , member t#refcount_t refcnt data_member_location(8) , member t#u8 flags data_member_location(12) } byte_size(16)
+s#ubuf_info_ops structure_type ubuf_info_ops { member pointer_type { subroutine_type ( formal_parameter pointer_type { s#sk_buff } , formal_parameter pointer_type { s#ubuf_info } , formal_parameter t#bool ) -> base_type void } complete data_member_location(0) , member pointer_type { subroutine_type ( formal_parameter pointer_type { s#sk_buff } , formal_parameter pointer_type { s#ubuf_info } ) -> base_type int byte_size(4) encoding(5) } link_skb data_member_location(8) } byte_size(16)
```

Then the problem is that struct ubuf_info now has a full definition in symtypes. The solution is to add a line to drivers/android/vendor_hooks.c:

```
ANDROID_KABI_DECLONLY(ubuf_info);
```

This instructs gendwarfksyms to treat the named type as undefined in the file.

A more complex possibility is that the new #include is itself in a header file. In this case you may need to distribute different sets of ANDROID_KABI_DECLONLY macro invocations across source files that indirectly pull in extra type definitions as some of them may already have had some of the type definitions.

For readability, place such macro invocations near the beginning of the source file.

Resolution for Android 15 and lower
Often, the fix is just hiding the new #include from genksyms.

```
#ifndef __GENKSYMS__
#include <linux/fwnode.h>
#endif
```

Otherwise, to identify the #include that causes the difference, follow these steps:

Open the header file that defines the symbol or data type having this difference. For example, edit include/linux/fwnode.h for the struct fwnode_handle.

Add the following code at the top of the header file:

```
#ifdef CRC_CATCH
#error "Included from here"
#endif
```

In the module's .c file that has a CRC mismatch, add the following as the first line before any of the #include lines.

```
#define CRC_CATCH 1
```

Compile your module. The resulting build-time error shows the chain of header file #include that led to this CRC mismatch. For example:

```
In file included from .../drivers/clk/XXX.c:16:
In file included from .../include/linux/of_device.h:5:
In file included from .../include/linux/cpu.h:17:
In file included from .../include/linux/node.h:18:
.../include/linux/device.h:16:2: error: "Included from here"
#error "Included from here"
```

One of the links in this chain of #include is due to a change made in your kernel, that's missing in the GKI kernel.

Case 2: Differences due to data type changes
If the CRC mismatch for a symbol or data type isn't due to a difference in visibility, then it's due to actual changes (additions, removals, or changes) in the data type itself.

For example, making the following change in your kernel causes several CRC mismatches as many symbols are indirectly affected by this type of change:

```
diff --git a/include/linux/iommu.h b/include/linux/iommu.h
 --- a/include/linux/iommu.h
 +++ b/include/linux/iommu.h
 @@ -259,7 +259,7 @@ struct iommu_ops {
   void (*iotlb_sync)(struct iommu_domain *domain);
   phys_addr_t (*iova_to_phys)(struct iommu_domain *domain, dma_addr_t iova);
   phys_addr_t (*iova_to_phys_hard)(struct iommu_domain *domain,
 -        dma_addr_t iova);
 +        dma_addr_t iova, unsigned long trans_flag);
   int (*add_device)(struct device *dev);
   void (*remove_device)(struct device *dev);
   struct iommu_group *(*device_group)(struct device *dev);
```

One CRC mismatch is for devm_of_platform_populate().

If you compare the .symtypes files for that symbol, it might look like this:

```
$ diff -u <GKI>/drivers/of/platform.symtypes <your kernel>/drivers/of/platform.symtypes
 --- <GKI>/drivers/of/platform.symtypes
 +++ <your kernel>/drivers/of/platform.symtypes
 @@ -399,7 +399,7 @@
 ...
 -s#iommu_ops structure_type iommu_ops { ... ; t#phy
 s_addr_t ( * iova_to_phys_hard ) ( s#iommu_domain * , t#dma_addr_t ) ; int
   ( * add_device ) ( s#device * ) ; ...
 +s#iommu_ops structure_type iommu_ops { ... ; t#phy
 s_addr_t ( * iova_to_phys_hard ) ( s#iommu_domain * , t#dma_addr_t , unsigned long ) ; int ( * add_device ) ( s#device * ) ; ...
```

To identify the changed type, follow these steps:

Find the definition of the symbol in the source code (usually in .h files).

For symbol differences between your kernel and the GKI kernel, find the commit by running the following command:

```
git blame
```

For deleted symbols (where a symbol is deleted in a tree and you also want to delete it in the other tree), you need to find the change that deleted the line. Use the following command on the tree where the line was deleted:

```
git log -S "copy paste of deleted line/word" -- <file where it was deleted>
```

Note: Don't copy and paste tabs.
Review the returned list of commits to locate the change or deletion. The first commit is probably the one you are searching for. If it isn't, go through the list until you find the commit.

After you identify the commit, either revert it in your kernel or update it to suppress the CRC change and upload it to ACK and get it merged. Each residual ABI break will need to be reviewed for safety and if necessary an allowed break can be recorded.

Prefer to consume existing padding
Some structures in GKI are padded to allow for their extension without breaking existing vendor modules. If an upstream commit (for example) adds a member to such a structure, then it may be possible to change it to consume some of the padding instead. This change is then hidden from CRC calculation.

The standardised, self-documenting macro ANDROID_KABI_RESERVE reserves a u64 worth of (aligned) space. It is used in place of a member declaration.

For example:

```
struct data {
       u64 handle;
       ANDROID_KABI_RESERVE(1);
       ANDROID_KABI_RESERVE(2);
};
```

Padding can be consumed, without affecting symbol CRCs, with ANDROID_KABI_USE (or ANDROID_KABI_USE2 or other variants that may be defined).

The member sekret is available as if it were directly declared, but the macro actually expands to an anonymous union member containing sekret as well as things used by gendwarfksyms to maintain symtype stability.

```
struct data {
       u64 handle;
       ANDROID_KABI_USE(1, void *sekret);
       ANDROID_KABI_RESERVE(2);
};
```

Resolution for Android 16 and higher
CRCs are calculated by gendwarfksyms which uses DWARF debugging information, thus supporting both C and Rust types. Resolution varies by the kind of type change. Here are some examples.

New or modified enumerators
Sometimes new enumerators are added and occasionally a MAX or similar enumerator value is also affected. These changes are safe if they don't

"escape" GKI or if we can be sure that vendor modules cannot care about their values.

For example:

```
 enum outcome {
       SUCCESS,
       FAILURE,
       RETRY,
+      TRY_HARDER,
       OUTCOME_LIMIT
 };
```

The addition of TRY_HARDER and the change to OUTCOME_LIMIT can hidden from CRC calculation with macro invocations at global scope:

```
ANDROID_KABI_ENUMERATOR_IGNORE(outcome, TRY_HARDER);
ANDROID_KABI_ENUMERATOR_VALUE(outcome, OUTCOME_LIMIT, 3);
```

For readability, place these just after the enum definition.

A new structure member occupying an existing hole
Due to alignment, there will be unused bytes between urgent and scratch.

```
        void *data;
        bool urgent;
+       bool retry;
        void *scratch;
```

No existing member offset or the size of the structure is affected by the addition of retry. However, it may affect symbol CRCs or the ABI representation or both.

This will hide it from CRC calculation:

```
        void *data;
        bool urgent;
+       ANDROID_KABI_IGNORE(1, bool retry);
        void *scratch_space;
```

The member retry is available as if it were directly declared, but the macro actually expands to an anonymous union member containing retry as well as things used by gendwarfksyms to maintain symtype stability.

Extension of a structure with new members
Members are sometimes added to the end of a structure. This does not affect offsets of existing members or affect existing users of the structure that only access it by pointer. The size of the structure affects its CRC and changes to this can be suppressed with an extra macro invocation at global scope, as follows:

```
struct data {
        u64 handle;
        u64 counter;
        ANDROID_KABI_IGNORE(1, void *sekret);
};

ANDROID_KABI_BYTE_SIZE(data, 16);
```

For readability, place this just after the struct definition.

All other changes to a type or the type of a symbol
Very occasionally, there may be changes that don't fall into one of previous categories, resulting in CRC changes that cannot be suppressed using the previous macros.

In these cases, the original symtypes description of a type or symbol can be supplied with an invocation of ANDROID_KABI_TYPE_STRING at global scope.

```
struct data {
        /* extensive changes */
};
```

ANDROID_KABI_TYPE_STRING("s#data", "original s#data symtypes definition");
For readability, place this just after the type or symbol definition.

Resolution for Android 15 and lower
Type and symbol type changes need to be hidden from genksyms. This can be done by controlling preprocessing with __GENKSYMS__.

Arbitrary code transformations can be expressed this way.

For example, to hide a new member occupying a hole in an existing structure:

```
struct parcel {
        void *data;
        bool urgent;
#ifndef __GENKSYMS__
        bool retry;
#endif
        void *scratch_space;
};
```

Got it! You've now explored:

- What Kernel ABI and KMI are
- Why ABI monitoring matters in Android kernels
- How to resolve CRC mismatches and ABI breakages using tooling and macros like `ANDROID_KABI_*`
- How to use symbol lists, stg files, and `Module.symvers` for tracking ABI stability

Let me know if you'd like a practical step-by-step example on resolving ABI breakages or need help setting up the environment to test these on a real kernel tree.

57. Refactor code to avoid ABI changes
Make every effort to avoid modifying the existing ABI. In many cases, you can refactor your code to remove changes that affect the ABI.

Refactoring struct field changes. If a change modifies the ABI for a debug feature, add an #ifdef around the fields (in the structs and source references) and make sure the CONFIG used for the #ifdef is disabled for the production defconfig and gki_defconfig. For an example of how a debug config can be added to a struct without breaking the ABI, refer to this patchset.

Refactoring features to not change the core kernel. If new features need to be added to ACK to support the partner modules, try to refactor the ABI part of the change to avoid modifying the kernel ABI. For an example of using the existing kernel ABI to add additional capabilities without changing the kernel ABI refer to aosp/1312213.

Fix a broken ABI on Android Gerrit
If you didn't intentionally break the kernel ABI, then you need to investigate, using the guidance provided by the ABI monitoring tooling. The most common causes of breakages are changed data structures and the associated symbol CRC changes, or due to config option changes that lead to any of the aforementioned. Begin by addressing the issues found by the tool.

You can reproduce the ABI findings locally, see Build the kernel and its ABI representation.

About Lint-1 labels
If you upload changes to a branch containing a frozen or finalized KMI, the changes must pass the AbiAnalyzer to verify that they don't affect the stable ABI in an incompatible way. During this process, the AbiAnalyzer looks for the ABI report that's created during the build (an extended build that performs the normal build and then some ABI extraction and comparison steps.

If the AbiAnalyzer finds a non-empty report it sets the Lint-1 label and the change is blocked from submittal until resolved; until the patchset receives a Lint+1 label.

Note: Patchsets can also receive a Lint-1 label by failing other analyzer checks.
Note: The analyzer also runs on the newer branches, but the results aren't valid until the KMI on a branch is finalized.
Update the kernel ABI
If modifying the ABI is unavoidable, then you must apply your code changes, the ABI representation, and symbol list to the ACK. To get Lint to remove the -1 and not break GKI compatibility, follow these steps:

Upload code changes to the ACK.

Wait to receive a Code-Review +2 for the patchset.

Update the reference ABI representation.

Merge your code changes and the ABI update change.

Note: If you haven't downloaded the ACK, refer to building Android kernels.
Upload ABI code changes to the ACK
Updating the ACK ABI depends on the type of change being made.

If an ABI change is related to a feature that affects CTS or VTS tests, the change can usually be cherry-picked to ACK as is. For example:

aosp/1289677 is needed for audio to work.
aosp/1295945 is needed for USB to work.
If an ABI change is for a feature that can be shared with the ACK, that change can be cherry-picked to ACK as is. For example, the following changes aren't needed for CTS or VTS test but are OK to be shared with ACK:

aosp/1250412 is a thermal feature change.
aosp/1288857 is an EXPORT_SYMBOL_GPL change.
If an ABI change introduces a new feature that doesn't need to be included in the ACK, you can introduce the symbols to ACK using a stub as as described in the following section.

Use stubs for ACK
Stubs must be necessary only for core kernel changes that don't benefit the ACK, such as performance and power changes. The following list details examples of stubs and partial cherry-picks in ACK for GKI.

Core-isolate feature stub (aosp/1284493). The capabilities in ACK isn't necessary, but the symbols need to be present in ACK for your modules to use these symbols.

Placeholder symbol for vendor module (aosp/1288860).

ABI-only cherry-pick of per-process mm event tracking feature (aosp/1288454). The original patch was cherry-picked to ACK and then trimmed to include only the necessary changes to resolve the ABI diff for task_struct and mm_event_count. This patch also updates the mm_event_type enum to contain the final members.

Partial cherry-pick of thermal struct ABI changes that required more than just adding the new ABI fields.

Patch aosp/1255544 resolved ABI differences between the partner kernel and ACK.

Patch aosp/1291018 fixed the functional issues found during GKI testing of the previous patch. The fix included initializing the sensor parameter struct to register multiple thermal zones to a single sensor.

CONFIG_NL80211_TESTMODE ABI changes (aosp/1344321). This patch added the necessary struct changes for ABI and made sure the additional fields didn't cause functional differences, enabling partners to include CONFIG_NL80211_TESTMODE in their production kernels and still maintain GKI compliance.

Caution: Partial cherry-picks of features can be tricky and must be done only for proprietary or hardware-specific features (or features that might break other partner modules in the ACK).
Enforce the KMI at runtime
The GKI kernels use the TRIM_UNUSED_KSYMS=y and UNUSED_KSYMS_WHITELIST=<union of all symbol lists> configuration options, which limit the exported symbols (such as symbols exported using EXPORT_SYMBOL_GPL()) to those listed on a symbol list. All other symbols are unexported, and loading a module requiring an unexported symbol is denied. This restriction is enforced at build time and missing entries are flagged.

For development purposes, you can use a GKI kernel build that doesn't include symbol trimming (meaning all usually exported symbols can be used). To locate these builds, look for the kernel_debug_aarch64 builds on ci.android.com.

Enforce the KMI using module versioning
The Generic Kernel Image (GKI) kernels use module versioning (CONFIG_MODVERSIONS) as an additional measure to enforce KMI compliance at runtime. Module versioning can cause cyclic redundancy check (CRC) mismatch failures at module load time if the expected KMI of a module doesn't match the vmlinux KMI. For example, the following is a typical failure that occurs at module load time due to a CRC mismatch for the symbol module_layout():

```
init: Loading module /lib/modules/kernel/.../XXX.ko with args ""
XXX: disagrees about version of symbol module_layout
init: Failed to insmod '/lib/modules/kernel/.../XXX.ko' with args ''
```

Uses of module versioning
Module versioning is useful for the following reasons:

Module versioning catches changes in data structure visibility. If modules change opaque data structures, that is, data structures that aren't part of the KMI, they break after future changes to the structure.

As an example, consider the fwnode field in struct device. This field MUST be opaque to modules so that they can't make changes to fields of device->fw_node or make assumptions about its size.

However, if a module includes <linux/fwnode.h> (directly or indirectly), then the fwnode field in the struct device is no longer opaque to it. The module can then make changes to device->fwnode->dev or device->fwnode->ops. This scenario is problematic for several reasons, stated as follows:

It can break assumptions the core kernel code is making about its internal data structures.

If a future kernel update changes the struct fwnode_handle (the data type of fwnode), then the module no longer works with the new kernel. Moreover, stgdiff won't show any differences because the module is breaking the KMI by directly manipulating internal data structures in ways that can't be captured by only inspecting the binary representation.

A current module is deemed KMI-incompatible when it is loaded at a later date by a new kernel that's incompatible. Module versioning adds a run-time check to avoid accidentally loading a module that isn't KMI-compatible with the kernel. This check prevents hard-to-debug runtime issues and kernel crashes that might result from an undetected incompatibility in the KMI.

Enabling module versioning prevents all these issues.

Check for CRC mismatches without booting the device
stgdiff compares and reports CRC mismatches between kernels along with other ABI differences.

In addition, a full kernel build with CONFIG_MODVERSIONS enabled generates a Module.symvers file as part of the normal build process. This file has one line for every symbol exported by the kernel (vmlinux) and the modules. Each line consists of the CRC value, symbol name, symbol namespace, the vmlinux or module name that's exporting the symbol, and the export type (for example, EXPORT_SYMBOL versus EXPORT_SYMBOL_GPL).

You can compare the Module.symvers files between the GKI build and your build to check for any CRC differences in the symbols exported by vmlinux. If there is a CRC value difference in any symbol exported by vmlinux and that symbol is used by one of the modules you load in your device, the module doesn't load.

If you don't have all the build artifacts, but do have the vmlinux files of the GKI kernel and your kernel, you can compare the CRC values for a specific symbol by running the following command on both the kernels and comparing the output:

```
nm <path to vmlinux>/vmlinux | grep __crc_<symbol name>
```

For example, the following command checks the CRC value for the module_layout symbol:

```
nm vmlinux | grep __crc_module_layout
0000000008663742 A __crc_module_layout
```

Resolve CRC mismatches

Note: The CRC calculation and its inputs changed significantly between Android 15 (genksyms) and Android 16 (gendwarfksyms). While the investigative steps required are similar, the actual resolution of CRC mismatches is very different.

Use the following steps to resolve a CRC mismatch when loading a module:

Build the GKI kernel and your device kernel using the --kbuild_symtypes option as shown in the following command:

```
tools/bazel run --kbuild_symtypes //common:kernel_aarch64_dist
```

This command generates a .symtypes file for each .o file. See KBUILD_SYMTYPES in Kleaf for details.

For Android 13 and lower build the GKI kernel and your device kernel by prepending KBUILD_SYMTYPES=1 to the command you use to build the kernel, as shown in the following command:

```
KBUILD_SYMTYPES=1 BUILD_CONFIG=common/build.config.gki.aarch64 build/build.sh
```

When using build_abi.sh, the KBUILD_SYMTYPES=1 flag is implicitly set already.

Find the .c file in which the symbol with CRC mismatch is exported, using the following command:

```
git -C common grep EXPORT_SYMBOL.*module_layout
kernel/module/version.c:EXPORT_SYMBOL(module_layout);
```

The .c file has a corresponding .symtypes file in the GKI, and your device kernel build artifacts. Locate the .symtypes file using the following commands:

```
cd bazel-bin/common/kernel_aarch64/symtypes
ls -1 kernel/module/version.symtypes
```

In Android 13 and lower, using the legacy build scripts, the location is likely to be either out/$BRANCH/common or out_abi/$BRANCH/common.

Each .symtypes file is a plain text file consisting of type and symbol descriptions:

Each line is of the form key description where the description can refer to other keys in the same file.

Keys like [s|u|e|t]#foo refer to [struct|union|enum|typedef] foo. For example:

```
t#bool typedef _Bool bool
```

Keys with no x# prefix are just symbol names. For example:

```
find_module s#module * find_module ( const char * )
```

Compare the two files and fix all the differences.

It is best to generate symtypes with a build just before the problematic change and then at the problematic change. Saving all the files means they can be compared in bulk.

For example,

```
for f in $(find good bad -name '*.symtypes' | sed -r 's;^(good|bad)/;;' | LANG=C sort -u); do
  diff -N -U0 --label good/"$f" --label bad/"$f" <(LANG=C sort good/"$f") <(LANG=C sort bad/"$f")
done
```

Otherise, just compare the specific files of interest.

Note: Any keys which are added (or removed) can be ignore for the purposes of investigation. We only care about keys whose descriptions have changed.

Note: In Android 16 and higher, the order of the keys isn't deterministic, so the files should be sorted before comparison. There are also minor differences in the description syntax.

Note: Preferably, enable line wrapping in your diff tool. For example, use :set wrap when using vimdff.

Case 1: Differences due to data type visibility

A new #include can pull a new type definition (say of struct foo) into a source file, In these cases, its description in the corresponding .symtypes file will change from an empty structure_type foo { } to a full definition.

Note: In Android 15 and lower, the old description was struct foo { UNKNOWN }.

This will affect all the CRCs of all symbols in the .symtypes file whose descriptions depend directly or indirectly on the type definition.

For example, adding the following line to the include/linux/device.h file in your kernel causes CRC mismatches, one of which is for module_layout():

```
 #include <linux/fwnode.h>
```

Comparing the module/version.symtypes for that symbol, exposes the following differences:

```
 $ diff -u <GKI>/kernel/module/version.symtypes <your kernel>/kernel/module/version.symtypes
  --- <GKI>/kernel/module/version.symtypes
  +++ <your kernel>/kernel/module/version.symtypes
  @@ -334,12 +334,15 @@
  ...
  -s#fwnode_handle structure_type fwnode_handle { }
  +s#fwnode_reference_args structure_type fwnode_reference_args { s#fwnode_handle * fwnode ; unsigned int nargs ; t#u64 args [ 8 ] ; }
  ...
```

If the GKI kernel has the full type definition, but your kernel is missing it (very unlikely), then merge the latest Android Common Kernel into your kernel so that you are using the latest GKI kernel base.

In most cases, the GKI kernel is missing the full type definition in .symtypes, but your kernel has it due to additional #include directives.

Resolution for Android 16 and higher

Make sure that the affected source file includes the Android KABI stabilisation header:

```
#include <linux/android_kabi.h>
```

For each affected type, add ANDROID_KABI_DECLONLY(name); at global scope to the affected source file.

For example, if the symtypes diff was this:

```
--- good/drivers/android/vendor_hooks.symtypes
+++ bad/drivers/android/vendor_hooks.symtypes
@@ -1051 +1051,2 @@
-s#ubuf_info structure_type ubuf_info { }
+s#ubuf_info structure_type ubuf_info { member pointer_type { const_type { s#ubuf_info_ops } } ops data_member_location(0) , member t#refcount_t refcnt data_member_location(8) , member t#u8 flags data_member_location(12) } byte_size(16)
+s#ubuf_info_ops structure_type ubuf_info_ops { member pointer_type { subroutine_type ( formal_parameter pointer_type { s#sk_buff } , formal_parameter pointer_type { s#ubuf_info } , formal_parameter t#bool ) -> base_type void } complete data_member_location(0) , member pointer_type { subroutine_type ( formal_parameter pointer_type { s#sk_buff } , formal_parameter pointer_type { s#ubuf_info } ) -> base_type int byte_size(4) encoding(5) } link_skb data_member_location(8) } byte_size(16)
```

Then the problem is that struct ubuf_info now has a full definition in symtypes. The solution is to add a line to drivers/android/vendor_hooks.c:

```
ANDROID_KABI_DECLONLY(ubuf_info);
```

This instructs gendwarfksyms to treat the named type as undefined in the file.

A more complex possibility is that the new #include is itself in a header file. In this case you may need to distribute different sets of ANDROID_KABI_DECLONLY macro invocations across source files that indirectly pull in extra type definitions as some of them may already have had some of the type definitions.

For readability, place such macro invocations near the beginning of the source file.

Resolution for Android 15 and lower
Often, the fix is just hiding the new #include from genksyms.

```
#ifndef __GENKSYMS__
#include <linux/fwnode.h>
#endif
```

Otherwise, to identify the #include that causes the difference, follow these steps:

Open the header file that defines the symbol or data type having this difference. For example, edit include/linux/fwnode.h for the struct fwnode_handle.

Add the following code at the top of the header file:

```
#ifdef CRC_CATCH
#error "Included from here"
#endif
```

In the module's .c file that has a CRC mismatch, add the following as the first line before any of the #include lines.

```
#define CRC_CATCH 1
```

Compile your module. The resulting build-time error shows the chain of header file #include that led to this CRC mismatch. For example:

```
In file included from .../drivers/clk/XXX.c:16:
In file included from .../include/linux/of_device.h:5:
In file included from .../include/linux/cpu.h:17:
In file included from .../include/linux/node.h:18:
.../include/linux/device.h:16:2: error: "Included from here"
#error "Included from here"
```

One of the links in this chain of #include is due to a change made in your kernel, that's missing in the GKI kernel.

Case 2: Differences due to data type changes
If the CRC mismatch for a symbol or data type isn't due to a difference in visibility, then it's due to actual changes (additions, removals, or changes) in the data type itself.

For example, making the following change in your kernel causes several CRC mismatches as many symbols are indirectly affected by this type of change:

```
diff --git a/include/linux/iommu.h b/include/linux/iommu.h
 --- a/include/linux/iommu.h
 +++ b/include/linux/iommu.h
 @@ -259,7 +259,7 @@ struct iommu_ops {
   void (*iotlb_sync)(struct iommu_domain *domain);
   phys_addr_t (*iova_to_phys)(struct iommu_domain *domain, dma_addr_t iova);
   phys_addr_t (*iova_to_phys_hard)(struct iommu_domain *domain,
 -        dma_addr_t iova);
 +        dma_addr_t iova, unsigned long trans_flag);
   int (*add_device)(struct device *dev);
   void (*remove_device)(struct device *dev);
   struct iommu_group *(*device_group)(struct device *dev);
```

One CRC mismatch is for devm_of_platform_populate().

If you compare the .symtypes files for that symbol, it might look like this:

```
 $ diff -u <GKI>/drivers/of/platform.symtypes <your kernel>/drivers/of/platform.symtypes
```

```
--- <GKI>/drivers/of/platform.symtypes
+++ <your kernel>/drivers/of/platform.symtypes
@@ -399,7 +399,7 @@
...
-s#iommu_ops structure_type iommu_ops { ... ; t#phy
s_addr_t ( * iova_to_phys_hard ) ( s#iommu_domain * , t#dma_addr_t ) ; int
  ( * add_device ) ( s#device * ) ; ...
+s#iommu_ops structure_type iommu_ops { ... ; t#phy
s_addr_t ( * iova_to_phys_hard ) ( s#iommu_domain * , t#dma_addr_t , unsigned long ) ; int ( * add_device ) ( s#device * ) ; ...
```

To identify the changed type, follow these steps:

Find the definition of the symbol in the source code (usually in .h files).

For symbol differences between your kernel and the GKI kernel, find the commit by running the following command:

```
git blame
```

For deleted symbols (where a symbol is deleted in a tree and you also want to delete it in the other tree), you need to find the change that deleted the line. Use the following command on the tree where the line was deleted:

```
git log -S "copy paste of deleted line/word" -- <file where it was deleted>
```

Note: Don't copy and paste tabs.

Review the returned list of commits to locate the change or deletion. The first commit is probably the one you are searching for. If it isn't, go through the list until you find the commit.

After you identify the commit, either revert it in your kernel or update it to suppress the CRC change and upload it to ACK and get it merged. Each residual ABI break will need to be reviewed for safety and if necessary an allowed break can be recorded.

Prefer to consume existing padding

Some structures in GKI are padded to allow for their extension without breaking existing vendor modules. If an upstream commit (for example) adds a member to such a structure, then it may be possible to change it to consume some of the padding instead. This change is then hidden from CRC calculation.

The standardised, self-documenting macro ANDROID_KABI_RESERVE reserves a u64 worth of (aligned) space. It is used in place of a member declaration.

For example:

```
struct data {
       u64 handle;
       ANDROID_KABI_RESERVE(1);
       ANDROID_KABI_RESERVE(2);
};
```

Padding can be consumed, without affecting symbol CRCs, with ANDROID_KABI_USE (or ANDROID_KABI_USE2 or other variants that may be defined).

The member sekret is available as if it were directly declared, but the macro actually expands to an anonymous union member containing sekret as well as things used by gendwarfksyms to maintain symtype stability.

```
struct data {
       u64 handle;
       ANDROID_KABI_USE(1, void *sekret);
       ANDROID_KABI_RESERVE(2);
};
```

Resolution for Android 16 and higher

CRCs are calculated by gendwarfksyms which uses DWARF debugging information, thus supporting both C and Rust types. Resolution varies by the kind of type change. Here are some examples.

New or modified enumerators

Sometimes new enumerators are added and occasionally a MAX or similar enumerator value is also affected. These changes are safe if they don't "escape" GKI or if we can be sure that vendor modules cannot care about their values.

For example:

```
 enum outcome {
      SUCCESS,
      FAILURE,
      RETRY,
+     TRY_HARDER,
      OUTCOME_LIMIT
 };
```

The addition of TRY_HARDER and the change to OUTCOME_LIMIT can hidden from CRC calculation with macro invocations at global scope:

```
ANDROID_KABI_ENUMERATOR_IGNORE(outcome, TRY_HARDER);
ANDROID_KABI_ENUMERATOR_VALUE(outcome, OUTCOME_LIMIT, 3);
```

For readability, place these just after the enum definition.

A new structure member occupying an existing hole
Due to alignment, there will be unused bytes between urgent and scratch.

```
       void *data;
       bool urgent;
+      bool retry;
       void *scratch;
```

No existing member offset or the size of the structure is affected by the addition of retry. However, it may affect symbol CRCs or the ABI representation or both.

This will hide it from CRC calculation:

```
       void *data;
       bool urgent;
+      ANDROID_KABI_IGNORE(1, bool retry);
       void *scratch_space;
```

The member retry is available as if it were directly declared, but the macro actually expands to an anonymous union member containing retry as well as things used by gendwarfksyms to maintain symtype stability.

Extension of a structure with new members
Members are sometimes added to the end of a structure. This does not affect offsets of existing members or affect existing users of the structure that only access it by pointer. The size of the structure affects its CRC and changes to this can be suppressed with an extra macro invocation at global scope, as follows:

```
struct data {
       u64 handle;
       u64 counter;
       ANDROID_KABI_IGNORE(1, void *sekret);
};

ANDROID_KABI_BYTE_SIZE(data, 16);
```

For readability, place this just after the struct definition.

All other changes to a type or the type of a symbol
Very occasionally, there may be changes that don't fall into one of previous categories, resulting in CRC changes that cannot be suppressed using the previous macros.

In these cases, the original symtypes description of a type or symbol can be supplied with an invocation of ANDROID_KABI_TYPE_STRING at global scope.

```
struct data {
       /* extensive changes */
};

ANDROID_KABI_TYPE_STRING("s#data", "original s#data symtypes definition");
```

For readability, place this just after the type or symbol definition.

Resolution for Android 15 and lower

Type and symbol type changes need to be hidden from genksyms. This can be done by controlling preprocessing with __GENKSYMS__.

Arbitrary code transformations can be expressed this way.

For example, to hide a new member occupying a hole in an existing structure:

```
struct parcel {
	void *data;
	bool urgent;
#ifndef __GENKSYMS__
	bool retry;
#endif
	void *scratch_space;
};
```

eXPLAIN IN SIMPLE WORDS

Update memory?

YesNo

Here's a **simple explanation** of the long technical document you shared. It's all about **how to change the Linux kernel code in Android without breaking compatibility with existing modules** (also called **preserving the Kernel Module Interface — KMI**).

🔧 What's the Problem?

When you change Linux kernel code (structs, functions, etc.), the **kernel ABI (Application Binary Interface)** can change.

- If the ABI changes, existing prebuilt modules may not work.
- This causes **"CRC mismatch"** errors when loading `.ko` kernel modules.
- Android strictly enforces ABI compatibility for **Generic Kernel Image (GKI)** to ensure **devices remain updatable**.

🧠 Goal

Make kernel changes without breaking the ABI or
properly handle and document the ABI change if it's unavoidable.

🛠️ Key Ideas & Tools

Concept	Simple Explanation
ABI	It's the binary agreement between kernel and modules — don't break it.
CRC	Unique "signature" of function/struct — changes if ABI changes.
Module.symvers	List of symbols exported by the kernel, with CRCs.
genksyms / gendwarfksyms	Tools that calculate CRCs based on struct/function definitions.
stgdiff / AbiAnalyzer	Tools that detect what ABI changed in your patch.
CONFIG_MODVERSIONS	Adds CRC to modules to detect version mismatch during load.
TRIM_UNUSED_KSYMS	Exports only needed symbols — blocks unused symbols.
.symtypes files	Internal kernel description of types used for CRCs in Android 16+.

👷 What Should You Do?

1. ✅ Avoid ABI Changes

- If you're adding a debug field in a struct:

```c
#ifdef CONFIG_MY_DEBUG_FEATURE
    int debug_field;
#endif
```

And make sure **`CONFIG_MY_DEBUG_FEATURE`** is off in production.

- If you're adding a new feature, try not to modify existing structs or interfaces. Use **existing symbols** or **helper layers**.

2. 📦 Use Macros to Hide Changes from ABI Calculators

For Android 16+ (with `gendwarfksyms`)

Use macros like:

Macro	Use
`ANDROID_KABI_IGNORE(1, my_field)`	Add a field but don't change CRC
`ANDROID_KABI_USE(1, my_field)`	Use a reserved field
`ANDROID_KABI_RESERVE(n)`	Reserve space for future
`ANDROID_KABI_DECLONLY(type)`	Pretend struct/enum is not visible
`ANDROID_KABI_ENUMERATOR_IGNORE(enum, value)`	Ignore new enum values
`ANDROID_KABI_BYTE_SIZE(type, size)`	Force same struct size
`ANDROID_KABI_TYPE_STRING("key", "old_value")`	Manually give old CRC version

For Android 15 and below (with `genksyms`)

Use:

```c
#ifndef __GENKSYMS__
    new_field;
#endif
```

This hides the field from ABI tools but still compiles.

3. 🔍 Debugging a CRC Mismatch

1. Build kernel with **`--kbuild_symtypes`** or **`KBUILD_SYMTYPES=1`**.
2. Check **`.symtypes`** files for changes.
3. Compare GKI version vs your version:

   ```sh
   diff good/foo.symtypes bad/foo.symtypes
   ```

4. If type details are different (e.g. full definition of struct), hide them with macros (**`ANDROID_KABI_DECLONLY`**).
5. If a new field changes a struct, hide it using **`ANDROID_KABI_IGNORE`** or consume a reserved space.

4. 📁 Update ABI Only If Needed

If ABI change is truly needed:

- Upload your kernel code.
- Get it reviewed.
- Regenerate ABI dump.
- Merge code **with ABI update patch**.
- GKI won't break as long as ABI dump is in sync.

5. 💡 Use Stubs for GKI

If your feature isn't needed in GKI but your module needs it:

- Add a **stub function** or **placeholder struct** in GKI so symbol is still present.
- This helps you keep **GKI compliance** while your vendor kernel has real logic.

📌 Summary: How to Avoid Breaking ABI

Case	What to Do
Add debug fields	Use `#ifdef CONFIG_DEBUG_FEATURE`
Add struct fields	Use `ANDROID_KABI_IGNORE` or `ANDROID_KABI_USE`
Add enums	Use `ANDROID_KABI_ENUMERATOR_IGNORE`
Hide struct changes	Use `ANDROID_KABI_DECLONLY()`
Hide includes	Wrap with `#ifndef __GENKSYMS__`
Extend structs	Reserve space with `ANDROID_KABI_RESERVE()`
Compare CRC	Use `nm`, `.symtypes`, or `Module.symvers`

Let me know if you'd like a **real example** of any of these techniques or a **diagram** explaining this visually.

58. Run ABI monitoring

bookmark_border
This page describes how to build Android kernel ABI representations and run ABI monitoring. It's applicable to Android 16 and higher. For lower versions, refer to ABI monitor previous kernel versions.

Also see the reference documentation for Kleaf: Support ABI monitoring (GKI) and Support ABI monitoring (Device).

Build the kernel and its ABI representation
After downloading GKI sources run the following command to build the GKI kernel and ABI artifacts:

tools/bazel run //common:kernel_aarch64_abi_dist
This command builds current ABI representation and copies it to $DIST_DIR/abi.stg along with the built kernel and modules. $DIST_DIR defaults to out_abi/kernel_aarch64_abi_dist/dist.

You can specify extra arguments for the ABI tooling at the end of the command after --. For example to change destination for ABI and build artifacts, you can use --destdir option:

tools/bazel run //common:kernel_aarch64_abi_dist -- --destdir=out/dist
Analyze the ABI differences between the build and a reference representation
The //common:kernel_aarch64_abi_dist target, executed in the preceding command, analyzes and reports any ABI differences found between the build and reference representation located at common/gki/aarch64/abi.stg (defined in BUILD.bazel). These differences are printed out at the end of the build, as shown in the following example:

INFO: From [stg] Comparing Kernel ABI @//common:kernel_aarch64_abi_diff:
INFO: ABI DIFFERENCES HAVE BEEN DETECTED!
The printed report comes from the build artifact located at $DIST_DIR/abi_stgdiff/abi.report.short along with reports in other formats.

Automation should use the build command's exit code, which is non-zero if differences are found.

Note that development phase branches, including android-mainline, have no reference ABI representation. Without it, //common:kernel_aarch64_abi_dist won't detect any differences.

Update the reference ABI representation
Any change that affects the kernel ABI, such as a symbol list update, needs to be reflected in the reference ABI representation (common/gki/aarch64/abi.stg, defined in BUILD.bazel). To do so you need to run the following command:

tools/bazel run //common:kernel_aarch64_abi_update
This command performs everything in the step Analyze the ABI differences and additionally updates the reference representation in the sources. The updated ABI can then be uploaded in the same commit as the change. Include the ABI differences from the report in $DIST_DIR/abi_stgdiff/abi.report.short in the commit message.

ABI monitoring and device targets
ABI monitoring only needs to be configured for core kernel build targets. Mixed build configurations (ones that define base_kernel) that compile directly with the GKI kernel only need to add support for tracking the device symbol list. The ABI definition should be updated using the GKI build.

Also see reference documentation for Kleaf: Support ABI monitoring (Device).

ABI monitor previous kernel versions (Android 15 and lower)
For previous kernel versions, ABI build and monitoring instructions differ as follows.

Android 15 and 14
The instructions are mostly the same as Android 16, except that the reference ABI representation is common/android/abi_gki_aarch64.stg and $DIST_DIR defaults to out_abi/kernel_aarch64/dist.

Android 13
Android 13 may be built either with Kleaf or with legacy build scripts.

For Kleaf, the instructions are the same as Android 14, except that the ABI format is XML and the reference ABI representation is common/android/abi_gki_aarch64.xml.

For legacy build scripts, the instructions are the same as Android 12.

Android 12 and lower
The ABI format is XML and the reference ABI representation is common/android/abi_gki_aarch64.xml

These kernels use build.sh instead of Kleaf. For ABI monitoring you should use build_abi.sh, which accepts the same environment variables to customize the build as build.sh. For example:

BUILD_CONFIG=common/build.config.gki.aarch64 build/build_abi.sh
This builds the kernel and extracts the ABI representation into the OUT_DIR (which is out_abi by default) subdirectory and is equivalent to the //common:kernel_aarch64_abi_dist target of Kleaf (see Build kernel and ABI artifacts).

The reference ABI representation is stored in android/abi_gki_aarch64.xml as defined by the ABI_DEFINITION variable in common/build.config.gki.aarch64.

If you need to update the kernel ABI representation, the most convenient way is to use --update and --print-report options:

BUILD_CONFIG=common/build.config.gki.aarch64 build/build_abi.sh --update --print-report
The --print-report prints the ABI differences between the file as it exists and a newly-generated ABI.

The --update option overwrites the reference ABI representation. It also does the symbol list update when using BUILD_CONFIG for a device with KMI_SYMBOL_LIST configured.

Here's a **simplified explanation** of how to **run ABI monitoring** for Android kernels (mostly Android 16+), and how it works across Android versions.

🔎 What is ABI Monitoring?

ABI (Application Binary Interface) defines **how your kernel modules interact with the kernel binary** (types, structures, symbols, etc). Any change in ABI may break existing modules.

ABI monitoring = watching for these ABI changes automatically.

✅ Android 16+ (Kleaf-based builds)

🛠️ Build Kernel and ABI Snapshot

Run this command:

```bash
tools/bazel run //common:kernel_aarch64_abi_dist
```

This will:

- Build the GKI kernel and modules.
- Generate the current ABI snapshot → **`abi.stg`**
- Compare this snapshot with the reference ABI file at:

  ```bash
  common/gki/aarch64/abi.stg
  ```

The result goes to:

```bash
out_abi/kernel_aarch64_abi_dist/dist/
```

If there are ABI changes, the report will say:

```makefile
INFO: ABI DIFFERENCES HAVE BEEN DETECTED!
```

🗂️ Output Report

Reports are generated under:

```bash
$DIST_DIR/abi_stgdiff/
```

You'll find:

- **`abi.report.short`** → Human-readable short diff
- **`abi.report.full`** → Full diff

📌 Want to Change the ABI?

If you **intentionally changed** ABI and it's valid:

Run:

```bash
tools/bazel run //common:kernel_aarch64_abi_update
```

This does:

- Same as **`abi_dist`**
- Updates the **reference ABI file `common/gki/aarch64/abi.stg`**

📌 Include the **`abi.report.short`** content in your **commit message**.

💡 Extra Options

To change output directory:

```bash
tools/bazel run //common:kernel_aarch64_abi_dist -- --destdir=out/dist
```

📱 ABI Monitoring for Device Kernels

- Device kernels **rely on GKI's ABI definition**.
- Device builds should track **device symbol list** only (not full ABI).
- ABI diffs should be **resolved in GKI**, not device kernels.

⏳ Android 15 and Below

🔄 Key Differences:

Android Version	ABI Format	Reference ABI Location	Build Tool
16+	**`.stg`**	**`common/gki/aarch64/`**	**`bazel`** (Kleaf)
14–15	**`.stg`**	**`common/android/`**	**`bazel`** (Kleaf)
13	**`.xml`**	**`common/android/`**	**`bazel`** or legacy
≤12	**`.xml`**	**`common/android/`**	**`build.sh`** or **`build_abi.sh`**

🧱 Legacy Build (Android 13 and below)

To monitor ABI with legacy tools:

```bash
BUILD_CONFIG=common/build.config.gki.aarch64 build/build_abi.sh
```

📁 ABI report is stored in:

```bash
out_abi/common/
```

🛠️ Update ABI (legacy)

If you intentionally changed ABI:

```bash
build/build_abi.sh --update --print-report
```

- **`--print-report`**: shows what changed
- **`--update`**: updates the reference ABI XML file

🚨 Important Notes

- **All GKI branches (like `android12-5.10`) have a frozen ABI** once finalized.
- ABI differences must be handled using proper tools (**`ANDROID_KABI_IGNORE`**, etc.) or by updating the ABI intentionally with review.
- Use **exit code** in CI to fail on unexpected ABI changes.
- For Android 16+, Kleaf tools are **mandatory**.

💬 Summary

Task	Command (Android 16+)
Build ABI & Check Diffs	**`tools/bazel run //common:kernel_aarch64_abi_dist`**
Update Reference ABI	**`tools/bazel run //common:kernel_aarch64_abi_update`**
Custom Output Dir	**`-- --destdir=out/dist`**
Old Devices (≤13)	Use **`build/build_abi.sh`**

Let me know if you want:

- 📊 A visual diagram of this process
- ✅ Steps for writing ABI-safe code
- 🧪 A working script to compare **`abi.stg`** files between two builds

59. what exactly is ABI explain with example

✅ What is ABI (Application Binary Interface)?

An **ABI** defines **how compiled binary code (e.g., kernel modules)** interacts with the **kernel** — it's like a *contract* between them.

It tells things like:

- How **data structures** look (size, layout, alignment).
- How **functions** are called (calling convention, return type, parameters).
- What **symbols** (functions or variables) are **available and visible** to external modules.
- Where things are located in **memory**.
- How **types** are defined and **aligned**.

> 🔥 **Changing the ABI breaks binary compatibility** — your `.ko` kernel module might not load anymore or crash at runtime.

🔧 Simple Analogy

Think of the ABI as a **socket shape** between a plug (module) and a socket (kernel). If you change the shape of the socket, old plugs won't fit.

🧪 Real-World Kernel Example

Suppose we have a struct in the kernel:

```c
// kernel headers (v1)
struct sensor_data {
    int temperature;
    int pressure;
};
```

A vendor module uses this struct:

```c
struct sensor_data *data = get_sensor_data();
printk("Temp: %d", data->temperature);
```

Now imagine kernel updates the struct:

```c
// kernel headers (v2) - ABI BREAKING CHANGE
struct sensor_data {
    int temperature;
    int pressure;
    int humidity;  // 🚨 New field added
};
```

Now, if the **module was compiled using v1**, but the kernel is now v2:

- The **size and layout** of the struct has changed.
- The module's binary still expects the **old layout**.
- Reading `data->temperature` might now point to the wrong field → crash.

🔴 **This is an ABI break.**

🧩 Symbol ABI Example

Suppose this kernel function is exported:

```c
// v1
int get_temperature(void);
EXPORT_SYMBOL(get_temperature);
```

If kernel changes it:

```c
// v2
long get_temperature(int sensor_id);  // changed return type & params
EXPORT_SYMBOL(get_temperature);
```

Then:

- Old modules that call `get_temperature()` will fail to link or crash.
- The **symbol signature has changed** → CRC mismatch.

This is also an **ABI break**.

📦 ABI is Not API

- **API** = source-level interface (headers, function names, etc.).
- **ABI** = binary-level interface (struct sizes, function signatures, memory layout).

  You can **change API** without breaking ABI (e.g., internal headers).
  But if you change **ABI**, old binaries might stop working.

✅ Why ABI Matters in Android Kernels?

- Android uses **Generic Kernel Image (GKI)** — all vendor modules must work with this GKI.
- GKI **freezes** the ABI, and modules must follow it exactly.
- Kernel uses **CRC checks** and **symbol lists** to **enforce ABI** at runtime.

🧪 Example ABI Break Detection

If kernel changes a symbol's type:

```bash
nm vmlinux | grep __crc_module_layout
00000000abcdef12 A __crc_module_layout  # CRC value in v1
```

If in new kernel it's:

```kotlin
00000000a1b2c3d4 A __crc_module_layout  # CRC changed → ABI break
```

Then kernel will **refuse to load** the old module.

💡 Summary

Feature	ABI Controls
Struct size & layout	✅
Function signature	✅
Symbol names & visibility	✅
Type alignment	✅
Memory offsets	✅

If your module binary assumes one thing and the kernel provides another, **ABI mismatch = crash, reject, or undefined behavior**.

Let me know if you want:

- 📊 Visual explanation of ABI vs API
- ✅ How to avoid ABI breaks in your kernel code
- 🛠️ Tools used to monitor ABI changes (`stgdiff`, `genksyms`, etc.)

60. Work with symbol lists

bookmark_border
To reduce the surface of symbols and types that need to be maintained as stable, the GKI kernel has capabilities to limit exported symbols to only those that are needed by modules. For externally compiled modules, you need to have a list of used symbols to allow them to be exported by the GKI kernel. For example, symbols used by modules for Cuttlefish are stored in gki/aarch64/symbols/virtual_device.

Add a target for the symbol list generation
Symbol lists are generated by the kernel_abi target. Add this target to the device BUILD.bazel with the following options:

name

Should be in the format of <kernel_build>_abi.

kernel_build

Should contain the name of the device kernel_build target.

You can also use the following options:

kernel_modules

List of the targets for out-of-tree modules. In-tree modules shouldn't be included here. Refer to Prepare in-tree modules for symbol extraction.

kmi_symbol_list_add_only

This option prevents the removal of unused symbols. Symbol removal is only permitted at specific times during KMI stabilization and is not allowed once the KMI is frozen.

This is also useful when you use the same symbol list for multiple different devices. This way it won't remove symbols used by device A but not device B.

module_grouping

If True or unspecified, then the symbol list groups symbols based on the kernel modules that reference the symbol. Otherwise the symbol list is a sorted list of symbols used by all the kernel modules.

See common-modules/virtual-device/BUILD.bazel for example:

```
kernel_abi(
    name = "virtual_device_aarch64_abi",
    kernel_build = ":virtual_device_aarch64",
    kernel_modules = [":virtual_device_aarch64_external_modules"],
    kmi_symbol_list_add_only = True,
)
```

Also see reference documentation on kernel_abi target in Kleaf.

Prepare in-tree modules for symbol extraction
To prepare in-tree modules for symbol extraction, list vendor-specific in-tree modules in a module_outs attribute of the kernel_build target. See _VIRT_COMMON_MODULES and its usage for an example. Don't include GKI modules in this list.

Configure these modules to be unsigned, otherwise the symbol list might be empty. To do so, add this line to your kernel config fragments:

```
# CONFIG_MODULE_SIG_ALL is not set
```

See common-modules/virtual-device/virtual_device_core.fragment for example.

Add a device symbol list to the device kernel build
Add attribute kmi_symbol_list to the kernel_build target defined in the device BUILD.bazel. The name of the symbol list should be in the format of //common:gki/<arch>/symbols/<device>.

Note: For Android 15 and lower, replace gki/<arch>/symbols/<device> with android/abi_gki_<arch>_<device>.

See common-modules/virtual-device/BUILD.bazel, for example:

```
kernel_build(
    name = "virtual_device_aarch64",
    base_kernel = "//common:kernel_aarch64",
    kmi_symbol_list = "//common:gki/aarch64/symbols/virtual_device",
    ...
    module_outs = _VIRT_COMMON_MODULES + _VIRT_AARCH64_MODULES,
)
```

Create and submit an initial symbol list
Create an empty symbol list at common/gki/<arch>/symbols/<device>. For the example above the command would be:

```
touch common/gki/aarch64/symbols/virtual_device
```

Add this file to additional_kmi_symbol_lists of the base GKI kernel build. For example, //common:gki/aarch64/symbols/virtual_device is added to the aarch64_additional_kmi_symbol_lists filegroup, declared in common/BUILD.bazel.

Update the device symbol list to fill the new symbol list and send it to the Android Common Kernel repository.

Update a device symbol list
All core kernel symbols used by modules in module_outs of kernel_build and kernel_modules of kernel_abi should be included in the symbol list. This can be done by running the kernel_abi target with the _update_symbol_list suffix. For example, the following command updates the symbol list for //common-modules/virtual-device:virtual_device_aarch64:

```
tools/bazel run //common-modules/virtual-device:virtual_device_aarch64_abi_update_symbol_list
```

Send a symbol list update to ACK
Send a patch with the symbol list change to the Android Common Kernel gerrit to make the new symbols part of the KMI.

The commit message should include a list of added or removed symbols. You can either write this list manually for a small symbol list update or use $DIST_DIR/abi_stgdiff/abi.report.short report after updating the reference ABI representation.

While updating the reference ABI representation before sending a symbol list update is not required, it may eliminate extra presubmit steps and make the change ready to submit faster. In any case, it is checked and updated if necessary during presubmit.

Work with symbol lists (Android 13)
Android 13 supports both Kleaf and legacy build scripts, so you can manage symbol lists either as described in the preceding sections or as described in the following section.

Work with symbol lists (Android 12 and lower)
Use build_abi.sh tool as follows:

```
BUILD_CONFIG=path/to/build.config.device build/build_abi.sh --update-symbol-list
```

In this example, build.config.device must include these configuration options:

vmlinux

Must be part of the FILES list. This can be done by including build.config.aarch64.

KMI_SYMBOL_LIST

Must be set and pointed at the KMI symbol list to update.

After updating the device symbol list you also need to reflect these changes in the GKI build (common/build.config.gki.aarch64):

Copy the updated symbol list to common/android/abi_gki_aarch64_<device>.

Check that android/abi_gki_aarch64_<device> is included in ADDITIONAL_KMI_SYMBOL_LISTS in common/build.config.gki.aarch64.

Send symbol list update to ACK.

Let’s break down **“Working with symbol lists in Android kernels”** into **simple words and examples** so you can confidently use it in real projects.

✅ What is a Kernel Symbol List?

A **symbol list** is a list of **kernel functions or variables** (called "symbols") that you allow to be accessed by **vendor modules**.

These are typically symbols exported using **`EXPORT_SYMBOL()`** or **`EXPORT_SYMBOL_GPL()`** from the core kernel.

> In Android's GKI model, **only listed symbols are exported**, to keep the ABI stable and surface small.

🧠 Why Use a Symbol List?

- GKI (Generic Kernel Image) is **shared across devices**.
- Vendor drivers/modules (like WiFi, GPU, sensors) must load without modifying the GKI.
- To do this, **they can only use symbols listed in the symbol list**.
- This helps ensure **KMI (Kernel Module Interface)** stability and compatibility across updates.

🧪 Example

Let's say your vendor kernel module uses **`platform_driver_register`**.

To make this symbol available to your module:

1. Add this symbol to your device-specific symbol list:

```nginx
platform_driver_register
```

2. Ensure your GKI build **exports** that symbol by checking:

```c
EXPORT_SYMBOL(platform_driver_register);
```

🔨 How to Work with Symbol Lists

Here's a simplified step-by-step flow with real commands.

Step 1: 📁 Create an empty symbol list for your device

```bash
touch common/gki/aarch64/symbols/your_device
```

Make sure this path matches the **`kmi_symbol_list`** in your **`BUILD.bazel`** file.

Step 2: 🛠️ Setup BUILD.bazel entries

In your device's **`BUILD.bazel`** file:

```python
kernel_build(
    name = "your_device_aarch64",
    base_kernel = "//common:kernel_aarch64",
    kmi_symbol_list = "//common:gki/aarch64/symbols/your_device",
    module_outs = ["your_vendor_module.ko"],
)

kernel_abi(
    name = "your_device_aarch64_abi",
    kernel_build = ":your_device_aarch64",
    kernel_modules = [":your_device_aarch64_external_modules"],
    kmi_symbol_list_add_only = True,
)
```

Step 3: 🪛 Build and Update the Symbol List

```bash
tools/bazel run //common-modules/your-device:your_device_aarch64_abi_update_symbol_list
```

This:

- Scans your kernel modules
- Finds which kernel symbols they use
- Adds them to the symbol list

Step 4: 📨 Send Patch to Android Common Kernel (ACK)

Now, commit and send the symbol list update to AOSP Gerrit:

```bash
git add common/gki/aarch64/symbols/your_device
git commit -m "Update KMI symbol list for your_device

Added:
 - platform_driver_register
 - of_find_compatible_node
"
repo upload .
```

📘 Tip: You can also copy contents of **`abi.report.short`** to the commit message for accuracy.

⚠️ Rules to Follow

Rule	Why
Only symbols used by modules go in the list	Keeps GKI minimal
Use **`kmi_symbol_list_add_only = True`**	Prevents deleting symbols used by other devices
Don't include in-tree GKI modules	They are managed separately
Disable module signing for symbol extraction	Use **`# CONFIG_MODULE_SIG_ALL is not set`**
Only update symbol list when needed	Overuse can bloat GKI

📜 Android 12 and Lower?

Use legacy script:

```bash
BUILD_CONFIG=path/to/build.config.device build/build_abi.sh --update-symbol-list
```

Then copy:

```bash
cp out_abi/.../abi_gki_aarch64_device common/android/abi_gki_aarch64_device
```

🧠 Summary

Task	Command or Config
Create list	**`touch common/gki/aarch64/symbols/<device>`**
Update list	**`tools/bazel run ..._abi_update_symbol_list`**
Use in build	**`kmi_symbol_list = "//common:gki/aarch64/symbols/<device>"`**
Submit	Patch to ACK with changes in list

Would you like a full working example for a fake module and symbol list?

61. Kernel modules overview

bookmark_border
There are two types of kernel modules: hardware agnostic GKI modules and hardware-specific vendor modules. This page provides an overview of both types of modules.

GKI modules
Generic kernel image (GKI) modules are used to deliver nonboot-required kernel capabilities separate from the generic core kernel. With GKI modules, you can choose specific kernel capabilities to use, often reducing kernel image size and runtime memory consumption. The reduction in size makes GKI well suited for Android Go devices and other resource-restricted form factors.

GKI modules also provide a mechanism to let vendors incorporate new upstream features after the KMI freeze milestone. Built-in code can't be replaced without building another image, whereas code delivered as a module can be replaced by another module.

GKI modules use the kernel's build time signing infrastructure to differentiate between GKI and other modules at run time. Unsigned modules are allowed to load as long as they only use symbols appearing on the allowlist or provided by other unsigned modules.

Note: For an overview of GKI-related terms, refer to Kernel overview.
There are two logical types of GKI modules: protected GKI module and unprotected GKI module.

Protected GKI module
A protected GKI module is delivered by Google, isn't restricted in any way, and behaves as though it is built with the kernel after loading. Additionally, protected GKI modules have the following characteristics:

Protected GKI modules have access to non-KMI kernel symbols that aren't available to vendor modules or unprotected GKI modules.
Protected GKI modules can export symbols that become part of the KMI surface as long as those symbols are cited in a symbol list.
Protected GKI modules can't be overridden by vendor modules.
A protected GKI module is the default class of GKI modules. All GKI modules are considered protected at the time of KMI freeze.

Unprotected GKI module
An unprotected GKI module can be overridden by a vendor module. After KMI freeze, a protected GKI module might be reclassified as unprotected if the GKI team decides that vendors need to override the default implementation with a version that includes new features from upstream Linux. On the next GKI release, unprotected modules are reclassified as protected after upstream code lands in an Android Common Kernel (ACK). Unprotected GKI modules have the following characteristics:

Unprotected GKI modules have the same access to exported symbols as vendor modules.
Unprotected GKI modules can't export symbols exported by protected GKI modules.
Unprotected GKI modules must preserve any KMI interfaces as though part of the core kernel.
Unprotected GKI modules can be overridden by vendor modules.
Vendor modules
A vendor module is delivered by partners to implement SoC and device-specific capabilities. Any existing kernel module that isn't delivered as part of the GKI kernel can be delivered as a vendor module.

Since one of the primary goals of the GKI project is to minimize hardware-specific code in the core kernel, vendors can expect that the GKI kernel won't include modules that are clearly managing their own hardware. For example, vendor ABC Inc. can expect that configs such as CONFIG_ABC_SOC_SUPPORT won't be enabled either as built-in or loadable GKI modules without their support.

If a kernel driver or framework exists in ACK, but isn't delivered as part of the GKI kernel, vendors can modify the driver and deliver it as a vendor module. Such modifications are discouraged for non vendor-specific modules because the same capabilities might be delivered with the GKI kernel in a future release. When the GKI kernel contains capabilities provided by a vendor module, the vendor module won't load. For example, CONFIG_GREYBUS isn't set for GKI in Android 11, so vendors may deliver greybus vendor modules. However, CONFIG_GREYBUS might be enabled as a GKI built-in or module in Android 12, in which case greybus vendor modules won't be loaded. A best practice is to use the upstream version of non vendor-specific drivers if they are delivered as vendor modules.

You can deliver vendor modules in the vendor or the vendor_boot image. Modules required early in the boot process must be in vendor_boot. There is a boot-time cost associated with loading modules from vendor_boot.

Here's a **clear and concise overview of kernel modules** in Android, especially focusing on **GKI vs vendor modules**, with examples to help you understand when and why each is used.

✅ What Are Kernel Modules?

Kernel modules are pieces of code you can **load into the kernel at runtime**, instead of compiling them into the kernel image.

They're used for:

- Adding **device drivers** (e.g., WiFi, GPU)
- Extending **kernel functionality** (e.g., file systems, network protocols)
- Updating features **without rebuilding the whole kernel**

🧱 Types of Kernel Modules in Android

There are **two major types** in Android's **GKI (Generic Kernel Image)** architecture:

1. GKI Modules (Google-delivered)

> Modules that are **part of the generic Android kernel**, **hardware-agnostic**, and maintained by Google.

Benefits:

- Smaller kernel image
- Reduced RAM use (important for Android Go)
- Updatable modules even **after KMI freeze**
- Shared across all devices → easier maintenance

🔍 Two Subtypes of GKI Modules:

Type	Can be Overridden?	Access to Internal Symbols	Exports Symbols to Vendors
Protected	❌ No	✅ Yes	✅ Yes (via symbol list)
Unprotected	✅ Yes	❌ No	❌ No

Example:

- `dm-verity.ko`, `binder.ko` could be protected GKI modules.
- A new `usbcore.ko` module might be unprotected temporarily so vendors can patch it.

---

2. Vendor Modules

> Modules provided by **SoC or device vendors** (like Qualcomm, MediaTek, Samsung).

Characteristics:

- Deliver hardware-specific drivers: WiFi, Camera, Display, Sensors
- Not included in GKI
- Can depend **only on allowed (KMI) symbols**
- Cannot override protected GKI modules

Placement:

- Place in `vendor` or `vendor_boot` image
  - Put early-boot modules (e.g., display) in `vendor_boot`

Example:

- `qcom_camera.ko`, `mtk_wifi.ko`, `samsung_display_driver.ko`

🔄 GKI Module Lifecycle with KMI

Phase	Action
Before KMI Freeze	All GKI modules = **protected**
After KMI Freeze	Some modules can be reclassified to **unprotected** (allowing vendors to override them temporarily)
Next GKI Release	Updated modules may become protected again after merging upstream changes

🔐 Kernel Module Signing

- GKI modules are **signed** using kernel's internal signing mechanism.
- Vendor modules are **unsigned**.
- Only symbols in **KMI symbol list** are available to unsigned modules.

📝 If your vendor module uses a GKI-internal function not in the KMI, it will **fail to load**.

⚠️ Conflict Example

Let's say:

- In Android 11: **`CONFIG_GREYBUS`** = **disabled**
  → You (vendor) deliver your own **`greybus.ko`**
- In Android 12: **`CONFIG_GREYBUS`** = **enabled as GKI module**
  → Your **`greybus.ko`** won't load anymore (conflict)

📌 **Best Practice:** Use GKI-provided drivers if available, and follow upstream changes.

🧠 Summary Table

Feature	GKI Module (Protected)	GKI Module (Unprotected)	Vendor Module
Delivered by	Google	Google	Vendor
Hardware specific?	No	No	Yes
Can override core kernel?	No	Yes	N/A
Can be overridden?	No	Yes	N/A
Access to internal kernel symbols	Yes	No	No
Can export symbols?	Yes (with list)	No	No
Requires KMI compatibility?	Yes	Yes	Yes

👷 Real-World Example

Your SoC has a custom I2C controller.

- Google won't deliver this in GKI.
- You implement **`my_i2c_driver.ko`** as a **vendor module**.
- You check the required kernel symbols (**`i2c_add_driver`**, etc.) are exported via the KMI symbol list.
- You add your module to **`vendor_boot`** if needed early in boot.

Let me know if you'd like:

- A full working vendor module example
- How to write the **`Android.bp`** or **`BUILD.bazel`** for modules
- How to extract used symbols from a **`.ko`** file

62. Configure kernel features as GKI modules

bookmark_border
This page covers how to configure a new kernel feature as a GKI module or configure an existing built-in kernel feature as a GKI module.

Note: Changes that result in a new GKI module or that change the protection status of a GKI module must be approved by Google.
Configure a new feature as a GKI module
For the new feature, edit gki_defconfig and set the required kernel feature's config item from n to m (=m). Set this setting in both arch/arm64/configs/gki_defconfig and arch/x86/configs/gki_defconfig.

Add the KO (.ko) files generated for the feature to the COMMON_GKI_MODULES_LIST section of common/modules.bzl. Add the files in a sorted order. If you're unsure of all the files generated, the build fails and lists all the necessary KO files to be added to the list.

For Android 14, add the same set of KO files from step 2, sorted in ascending order for binary search at runtime, to common/android/gki_{ARCH}_protected_modules to designate the module as a protected GKI module.

For Android 14 and 15, update the list of exports so that it includes the newly-added exports in common/android/abi_gki_protected_exports_ARCHITECTURE. For example, to update the list, run tools/bazel run //common:kernel_aarch64_abi_update_protected_exports for aarch64.

Make sure newly added KO files from step 2 are copied to the kernel's out/<androidX-Y.Z>/dist/system_dlkm.img and out/androidX-Y.Z/dist/system_dlkm_staging_archive.tar.gz. Modules in the system_dlkm_staging_archive.tar.gz archive can be used as input to generate the system_dlkm.img in the platform build.

Submit your changes for review. GKI modules are an Android-only kernel feature, so module conversion patches aren't required to be submitted upstream. However, you must follow other guidelines to submit Android Common Kernel (ACK) patches.

Configure a kernel built-in feature as a GKI module
For an existing built-in kernel feature, edit gki_defconfig and set the required kernel feature's config item from y to m (=m). Set this setting in both arch/arm64/configs/gki_defconfig and arch/x86/configs/gki_defconfig.

Add the KO (.ko) files generated for the feature to the COMMON_GKI_MODULES_LIST section of common/modules.bzl. Add the files in a sorted order. If you're unsure of all the files generated, the build fails and lists all the necessary KO files to be added to the list.

For Android 14, add the same set of KO files from step 2, sorted in ascending order for binary search at runtime, to common/android/gki_{ARCH}_protected_modules to designate the module as a protected GKI module.

For Android 14 and 15, update the list of exports that are protected to include ones from the newly added module in common/android/abi_gki_protected_exports_{ARCH} using tools/bazel run //common:kernel_aarch64_abi_update_protected_exports for aarch64.

Make sure newly converted module KO files from step 2 are copied to the kernel's out/<androidX-Y.Z>/dist/system_dlkm.img and out/androidX-Y.Z/dist/system_dlkm_staging_archive.tar.gz. Modules in the system_dlkm_staging_archive.tar.gz archive can be used as input to generate the system_dlkm.img in the platform build.

Submit your changes for review. GKI modules are an Android-only kernel feature, so module conversion patches aren't required to be submitted upstream. However, you must follow the other guidelines to submit Android Common Kernel (ACK) patches.

Convert a protected GKI module to unprotected
For Android 15 and higher, add the module being converted from protected to unprotected to the COMMON_UNPROTECTED_MODULES_LIST list in common/modules.bzl file.

For Android 14, remove the module being converted from protected to unprotected from the list of protected modules at common/android/gki_protected_modules.

For Android 14 and 15, update the list of exports that are protected to exclude ones from the newly converted unprotected module in the common/android/abi_gki_protected_exports_{ARCH} using tools/bazel run //common:kernel_aarch64_abi_update_protected_exports for aarch64.

Submit your changes for review. GKI modules are an Android-only kernel feature, so module conversion patches aren't required to be submitted upstream. However, you must follow the other guidelines to submit Android Common Kernel (ACK) patches.

GKI modules symbol violation resolution quick guide
When unsigned modules violate the symbol protection in place for GKI modules, two types of errors may be encountered during module loading, resulting in failure.

1. Unsigned module using the protected symbol
Error:

module: Protected symbol: some_kernel_function (err -13)

Cause:

The module.ko file is an unsigned vendor module and attempts to resolve the GKI module exported symbol some_kernel_function during loading, without being listed in the vendor symbol list.

Resolution:

If module.ko is not a protected GKI module, updating the symbol list will resolve the error by including some_kernel_function in the vendor symbol list. Alternatively, use the GKI version of module.ko.

2. Unsigned module exporting the protected symbol
Error:

module: exports protected symbol some_kernel_function

Cause:

The module exporting the some_kernel_function is a protected GKI module, and module.ko is likely an unsigned custom version of that module. When module.ko tries to export some_kernel_function, which can only be exported by a signed GKI module, loading fails with this message.

Resolution:

This can be corrected by using the GKI version of the module that exports some_kernel_function, if the unsigned module is a custom version.

Here's a **simple and complete explanation** of how to configure **kernel features as GKI modules**, **convert built-in features to modules**, and how **symbol protection works**—along with **real-world context** and **error resolution tips**.

🔧 What does it mean to "configure a kernel feature as a GKI module"?

When you enable a kernel feature as a **GKI module**:

- That feature is **built as a `.ko` file** (loadable module)
- It's **not statically linked** into the kernel image (`vmlinux`)
- Android can **load/unload it dynamically**
- It becomes part of the **Generic Kernel Image (GKI)**, shared across devices

---

✅ Steps to Enable a New Kernel Feature as GKI Module

Suppose you want to enable `CONFIG_EXAMPLE_DRIVER` as a GKI module.

1. Edit `gki_defconfig`

In both of these files:

- `arch/arm64/configs/gki_defconfig`
- `arch/x86/configs/gki_defconfig`

Set:

```make
CONFIG_EXAMPLE_DRIVER=m
```

This means: build as a module (`.ko`), not built-in (`=y`).

---

2. Add `.ko` files to the GKI module list

Open **`common/modules.bzl`** and add the **`.ko`** file to:

```python
COMMON_GKI_MODULES_LIST = [
    ...
    "example_driver.ko",
]
```

✅ Sort alphabetically.

🧠 Tip: If you miss any related **`.ko`**, Bazel build will fail and tell you what to add.

---

3. (Android 14+) Mark it as a Protected Module

Add **`example_driver.ko`** to:

```text
common/android/gki_aarch64_protected_modules
```

⚠️ Sorted list (binary search friendly).

This tells Android: "This module is part of the official protected GKI set."

---

4. Update protected symbol exports

Run:

```bash
tools/bazel run //common:kernel_aarch64_abi_update_protected_exports
```

This updates the file:

```text
common/android/abi_gki_protected_exports_aarch64
```

Which lists which **symbols can be accessed/exported** by protected GKI modules.

---

5. Verify Output `.ko` in dist archive

Ensure that **`example_driver.ko`** is packed into:

- **`out/androidX-Y.Z/dist/system_dlkm.img`**
- **`out/androidX-Y.Z/dist/system_dlkm_staging_archive.tar.gz`**

These are the modules delivered with the Android platform system.

---

6. Submit for Review

GKI module changes:

- **Do not need to be submitted upstream**
- But you **must follow ACK (Android Common Kernel) submission guidelines**

🔁 Converting Built-in Feature to GKI Module

Let's say **`CONFIG_OLD_FEATURE=y`** is built-in.

To convert it:

1. Change to `=m` in both `gki_defconfig` files
2. Add `.ko` to `COMMON_GKI_MODULES_LIST`
3. Mark it protected (if needed)
4. Update protected exports
5. Validate `.ko` is included in `system_dlkm.img`
6. Submit patches

🔃 Convert a Protected GKI Module → Unprotected

Why? To allow vendors to override it (temporarily).

1. **Add to** `COMMON_UNPROTECTED_MODULES_LIST` in `common/modules.bzl`
2. **Remove** it from `common/android/gki_protected_modules` (Android 14 only)
3. Run:

   ```bash
   tools/bazel run //common:kernel_aarch64_abi_update_protected_exports
   ```

   (removes its symbols from the protected exports list)
4. Submit for review

🔐 Symbol Violation Errors and Fixes

❌ Error 1: Vendor module uses protected symbol

```javascript
module: Protected symbol: some_kernel_function (err -13)
```

Cause:

- Your **unsigned vendor module** depends on a symbol only exposed to **protected GKI modules**

Fix:

- Add `some_kernel_function` to **vendor KMI symbol list**
  (if allowed)
- Or use **GKI-provided module.ko** that already exports it

❌ Error 2: Vendor module tries to export a protected symbol

```java
module: exports protected symbol some_kernel_function
```

Cause:

- Vendor module.ko tries to expose a symbol **only Google-signed GKI modules can export**

Fix:

- Use official GKI `.ko` file
- Don't try to override that module

🧠 Real Example

Let’s say you're enabling the NTFS file system as a GKI module:

1. In **`gki_defconfig`**:

   ```make
   CONFIG_NTFS_FS=m
   ```

2. Add:

   ```python
   "ntfs.ko",
   ```

   to **`COMMON_GKI_MODULES_LIST`**

3. Add **`ntfs.ko`** to **`gki_aarch64_protected_modules`**
4. Run:

   ```bash
   tools/bazel run //common:kernel_aarch64_abi_update_protected_exports
   ```

5. Ensure **`ntfs.ko`** appears in **`system_dlkm.img`**
6. Submit patches to ACK

⚙️ Summary Flow

```text
Feature =m → Add .ko to modules.bzl → Mark protected → Update exports list
→ Ensure .ko in dlkm archive → Submit patch to ACK
```

Let me know if you want:

- Example patch files
- How to extract missing **`.ko`** from a build
- Step-by-step to write the symbol list for a vendor module

63. Vendor module guidelines

bookmark_border
Use the following guidelines to increase the robustness and reliability of your vendor modules. Many guidelines, when followed, can help make it easier to determine the correct module load order and the order in which drivers must probe for devices.

Note: Guidelines are listed in the order of importance and reflect the frequency of errors found in vendor modules.
A module can be a library or a driver.

Library modules are libraries that provide APIs for other modules to use. Such modules typically aren't hardware-specific. Examples of library modules include an AES encryption module, the remoteproc framework that's compiled as a module, and a logbuffer module. The module code in module_init() runs to set up data structures, but no other code runs unless triggered by an external module.

Driver modules are drivers that probe for or bind to a specific type of device. Such modules are hardware-specific. Examples of driver modules include UART, PCIe, and video encoder hardware. Driver modules activate only when their associated device is present on the system.

If the device isn't present, the only module code that runs is the module_init() code that registers the driver with the driver core framework.

If the device is present and the driver successfully probes for or binds to that device, other module code might run.

Use module init and exit correctly
Driver modules must register a driver in module_init() and unregister a driver in module_exit(). One way to enforce these restrictions is to use wrapper macros, which avoids the direct use of module_init(), *_initcall(), or module_exit() macros.

For modules that can be unloaded, use module_subsystem_driver(). Examples: module_platform_driver(), module_i2c_driver(), and module_pci_driver().

For modules that can't be unloaded, use builtin_subsystem_driver() Examples: builtin_platform_driver(), builtin_i2c_driver(), and builtin_pci_driver().

Note: You can build drivers with builtin_subsystem_driver() as modules, but you can't unload these drivers after they're loaded successfully.
Some driver modules use module_init() and module_exit() because they register more than one driver. For a driver module that uses module_init() and module_exit() to register multiple drivers, try to combine the drivers into a single driver. For example, you could differentiate using the compatible string or the aux data of the device instead of registering separate drivers. Alternatively, you could split the driver module into two modules.

Init and exit function exceptions
Library modules don't register drivers and are exempt from restrictions on module_init() and module_exit() as they might need these functions to set up data structures, work queues, or kernel threads.

Use the MODULE_DEVICE_TABLE macro
Driver modules must include the MODULE_DEVICE_TABLE macro, which allows the user space to determine the devices supported by a driver module before loading the module. Android can use this data to optimize module loading, such as to avoid loading modules for devices that aren't present in the system. For examples on using the macro, refer to the upstream code.

Avoid CRC mismatches due to forward-declared data types
Don't include header files to get visibility into forward-declared data types. Some structs, unions, and other data types defined in a header file (header-A.h) can be forward declared in a different header file (header-B.h) that typically uses pointers to those data types. This code pattern means that the kernel is intentionally trying to keep the data structure private to the users of header-B.h.

Users of header-B.h shouldn't include header-A.h to directly access the internals of these forward-declared data structures. Doing so causes CONFIG_MODVERSIONS CRC mismatch issues (which generates ABI compliance issues) when a different kernel (such as the GKI kernel) attempts to load the module.

For example, struct fwnode_handle is defined in include/linux/fwnode.h, but is forward declared as struct fwnode_handle; in include/linux/device.h because the kernel is trying to keep the details of struct fwnode_handle private from the users of include/linux/device.h. In this scenario, don't add #include <linux/fwnode.h> in a module to gain access to members of struct fwnode_handle. Any design in which you have to include such header files indicates a bad design pattern.

Don't directly access core kernel structures
Directly accessing or modifying core kernel data structures can lead to undesirable behavior, including memory leaks, crashes, and broken compatibility with future kernel releases. A data structure is a core kernel data structure when it meets any of the following conditions:

The data structure is defined under KERNEL-DIR/include/. For example, struct device and struct dev_links_info. Data structures defined in include/linux/soc are exempted.

The data structure is allocated or initialized by the module but is made visible to the kernel by being passed, indirectly (through a pointer in a struct) or directly, as input in a function exported by the kernel. For example, a cpufreq driver module initializes the struct cpufreq_driver and then passes it as input to cpufreq_register_driver(). After this point, the cpufreq driver module shouldn't modify struct cpufreq_driver directly because calling cpufreq_register_driver() makes struct cpufreq_driver visible to the kernel.

The data structure isn't initialized by your module. For example, struct regulator_dev returned by regulator_register().

Access core kernel data structures only through functions exported by the kernel or through parameters explicitly passed as input to vendor hooks. If you don't have an API or vendor hook to modify parts of a core kernel data structure, it's probably intentional and you shouldn't modify the data structure from modules. For example, don't modify any fields inside struct device or struct device.links.

To modify device.devres_head, use a devm_*() function such as devm_clk_get(), devm_regulator_get(), or devm_kzalloc().

To modify fields inside struct device.links, use a device link API such as device_link_add() or device_link_del().

Don't parse devicetree nodes with compatible property
If a device tree (DT) node has a compatible property, a struct device is allocated for it automatically or when of_platform_populate() is called on the parent DT node (typically by the device driver of the parent device). The default expectation (except for some devices initialized early for the scheduler) is that a DT node with a compatible property has a struct device and a matching device driver. All other exceptions are already handled by the upstream code.

In addition, fw_devlink (previously called of_devlink) considers DT nodes with the compatible property to be devices with an allocated struct device that's probed by a driver. If a DT node has a compatible property but the allocated struct device isn't probed, fw_devlink could block its consumer devices from probing or could block sync_state() calls from being called for its supplier devices.

If your driver uses an of_find_*() function (such as of_find_node_by_name() or of_find_compatible_node()) to directly find a DT node that has a compatible property and then parse that DT node, fix the module by writing a device driver that can probe the device or remove the compatible property (possible only if it hasn't been upstreamed). To discuss alternatives, reach out to the Android Kernel Team at kernel-team@android.com and be prepared to justify your use cases.

Note: The use of fw_devlink is mandatory for GKI.
Use DT phandles to look up suppliers
Refer to a supplier using a phandle (a reference or pointer to a DT node) in DT whenever possible. Using standard DT bindings and phandles to refer to suppliers enables fw_devlink (previously of_devlink) to automatically determine inter-device dependencies by parsing the DT at runtime. The kernel can then automatically probe devices in the correct order, removing the need for module load ordering or MODULE_SOFTDEP().

Note: The following examples specify regulators, but the same guideline applies to supplier DT bindings such as clocks, iio-channels, interrupts, -gpios, pinctrl-, dmas, iommu, phys, extcon, and nvmem-cells.
Legacy scenario (no DT support in ARM kernel)
Previously, before DT support was added to ARM kernels, consumers such as touch devices looked up suppliers such as regulators using globally unique strings. For example, the ACME PMIC driver could register or advertise multiple regulators (such as acme-pmic-ldo1 to acme-pmic-ldo10) and a touch driver could look up a regulator using regulator_get(dev, "acme-pmic-ldo10"). However, on a different board, the LDO8 might supply the touch device, creating a cumbersome system where the same touch driver needs to determine the correct look-up string for the regulator for each board that the touch device is used in.

Current scenario (DT support in ARM kernel)
After DT support was added to ARM kernels, consumers can identify suppliers in the DT by referring to the supplier's device tree node using a phandle. Consumers can also name the resource based on what it's used for instead of who supplies it. For example, the touch driver from the previous example could use regulator_get(dev, "core") and regulator_get(dev, "sensor") to get the suppliers that power the touch device's core and sensor. The associated DT for such a device is similar to the following code sample:

```
touch-device {
  compatible = "fizz,touch";
  ...
  core-supply = <&acme_pmic_ldo4>;
  sensor-supply = <&acme_pmic_ldo10>;
};

acme-pmic {
  compatible = "acme,super-pmic";
  ...
  acme_pmic_ldo4: ldo4 {
```

```
    ...
  };
  ...
  acme_pmic_ldo10: ldo10 {
    ...
  };
};
```

Note: *-supply is the standardized DT binding (or property) for referring to regulators that supply a device.

Worst-of-both-worlds scenario

Some drivers ported from older kernels include legacy behavior in the DT that takes the worst part of the legacy scheme and forces it on the newer scheme that's meant to make things easier. In such drivers, the consumer driver reads the string to use for lookup using a device-specific DT property, the supplier uses another supplier-specific property to define the name to be used for registering the supplier resource, then the consumer and supplier continue using the same old scheme of using strings to look up the supplier. In this worst-of-both-worlds scenario:

The touch driver uses code similar to the following code:

```
str = of_property_read(np, "fizz,core-regulator");
core_reg = regulator_get(dev, str);
str = of_property_read(np, "fizz,sensor-regulator");
sensor_reg = regulator_get(dev, str);
```

The DT uses code similar to the following:

```
touch-device {
  compatible = "fizz,touch";
  ...
  fizz,core-regulator = "acme-pmic-ldo4";
  fizz,sensor-regulator = "acme-pmic-ldo4";
};
acme-pmic {
  compatible = "acme,super-pmic";
  ...
  ldo4 {
    regulator-name = "acme-pmic-ldo4"
    ...
  };
  ...
  acme_pmic_ldo10: ldo10 {
    ...
    regulator-name = "acme-pmic-ldo10"
  };
};
```

Note: If your driver implements this worst-of-both-worlds scenario, fix it immediately to use phandles to look up suppliers.

Don't modify framework API errors

Framework APIs, such as regulator, clocks, irq, gpio, phys, and extcon, return -EPROBE_DEFER as an error return value to indicate that a device is attempting to probe but can't at this time, and the kernel should reattempt the probe later. To ensure that your device's .probe() function fails as expected in such cases, don't replace or remap the error value. Replacing or remapping the error value might cause -EPROBE_DEFER to be dropped and result in your device never getting probed.

Use devm_*() API variants

When the device acquires a resource using a devm_*() API, the resource is automatically released by the kernel if the device fails to probe, or probes successfully and is later unbound. This capability makes the error handling code in the probe() function cleaner because it doesn't require goto jumps to release the resources acquired by devm_*() and simplifies driver unbinding operations.

Handle device-driver unbinding

Be intentional about unbinding device drivers and don't leave the unbinding undefined because undefined doesn't imply disallowed. You must either fully implement device-driver unbinding or explicitly disable device-driver unbinding.

Note: There are a few cases where not doing anything explicitly is OK because the kernel handles any needed operations.

Implement device-driver unbinding

When choosing to fully implement device-driver unbinding, unbind device drivers cleanly to avoid memory or resource leaks and security issues. You can bind a device to a driver by calling a driver's probe() function and unbind a device by calling the driver's remove() function. If no remove() function exists, the kernel can still unbind the device; the driver core assumes that no clean up work is needed by the driver when it unbinds from the device. A driver that's unbound from a device doesn't need to do any explicit clean up work when both of the following are true:

All resources acquired by a driver's probe() function are through devm_*() APIs.

The hardware device doesn't need a shutdown or quiescing sequence.

In this situation, the driver core handles releasing all resources acquired through devm_*() APIs. If either of the preceding statements is untrue, the driver needs to perform cleanup (release resources and shut down or quiesce the hardware) when it unbinds from a device. To ensure that a device can unbind a driver module cleanly, use one of the following options:

If the hardware doesn't need a shutdown or quiescing sequence, change the device module to acquire resources using devm_*() APIs.

Implement the remove() driver operation in the same struct as the probe() function, then do the clean up steps using the remove() function.

Explicitly disable device-driver unbinding (not recommended)
When choosing to explicitly disable device-driver unbinding, you need to disallow unbinding and disallow module unloading.

To disallow unbinding, set the suppress_bind_attrs flag to true in the driver's struct device_driver; this setting prevents the bind and unbind files from showing in the driver's sysfs directory. The unbind file is what allows user space to trigger the unbinding of a driver from its device.

To disallow module unloading, ensure the module has [permanent] in lsmod. By not using module_exit() or module_XXX_driver(), the module is marked as [permanent].

Don't load firmware from within the probe function
Driver shouldn't load firmware from within the .probe() function as they might not have access to the firmware if the driver probes before the flash or permanent storage based file system is mounted. In such cases, the request_firmware*() API might block for a long time and then fail, which can slow the boot process unnecessarily. Instead, defer the loading of the firmware to when a client starts using the device. For example, a display driver could load the firmware when the display device is opened.

Using .probe() to load firmware might be OK in some cases, such as in a clock driver that needs firmware to function but the device isn't exposed to user space. Other appropriate use cases are possible.

Implement asynchronous probing
Support and use asynchronous probing to take advantage of future enhancements, such as parallel module loading or device probing to speed up boot time, that might be added to Android in future releases. Driver modules that don't use asynchronous probing could reduce the effectiveness of such optimizations.

To mark a driver as supporting and preferring asynchronous probing, set the probe_type field in the driver's struct device_driver member. The following example shows such support enabled for a platform driver:

```
static struct platform_driver acme_driver = {
        .probe          = acme_probe,
        ...
        .driver         = {
                .name   = "acme",
                ...
                .probe_type = PROBE_PREFER_ASYNCHRONOUS,
        },
};
```

Making a driver work with asynchronous probing doesn't require special code. However, keep the following in mind when adding asynchronous probing support.

Don't make assumptions about previously probed dependencies. Check directly or indirectly (most framework calls) and return -EPROBE_DEFER if one or more suppliers aren't ready yet.

If you add child devices in a parent device's probe function, don't assume the child devices is probed immediately.

If a probe fails, perform proper error handling and clean up (see Use devm_*() API variants).

Don't use MODULE_SOFTDEP to order device probes
The MODULE_SOFTDEP() function isn't a reliable solution for guaranteeing the order of device probes and mustn't be used for the following reasons.

Deferred probe. When a module loads, the device probe might be deferred because one of its suppliers isn't ready. This can lead to a mismatch between the module load order and the device probe order.

One driver, many devices. A driver module can manage a specific device type. If the system includes more than one instance of a device type and those devices each have a different probe order requirement, you can't respect those requirements using module load ordering.

Asynchronous probing. Driver modules that perform asynchronous probing don't immediately probe a device when the module is loaded. Instead, a parallel thread handles device probing, which can lead to a mismatch between the module load order and the device probe order. For example, when an I2C main driver module performs asynchronous probing and a touch driver module depends on the PMIC that's on the I2C bus, even if the touch driver and the PMIC driver load in the correct order, the touch driver's probe might be attempted before the PMIC driver probe.

If you have driver modules using the MODULE_SOFTDEP() function, fix them so they don't use that function. To help you, the Android team has upstreamed changes that enable the kernel to handle ordering problems without using MODULE_SOFTDEP(). Specifically, you can use fw_devlink to ensure probe ordering and (after all consumers of a device have probed) use the sync_state() callback to perform any necessary tasks.

Use #if IS_ENABLED() instead of #ifdef for configurations
Use #if IS_ENABLED(CONFIG_XXX) instead of #ifdef CONFIG_XXX to ensure that the code inside the #if block continues to compile if the config changes to a tristate config in the future. The differences are as follows:

#if IS_ENABLED(CONFIG_XXX) evaluates to true when CONFIG_XXX is set to module (=m) or built-in (=y).

#ifdef CONFIG_XXX evaluates to true when CONFIG_XXX is set to built-in (=y) , but doesn't when CONFIG_XXX is set to module (=m). Use this only when you're certain you want to do the same thing when the config is set to module or is disabled.

Note: Use #if IS_ENABLED(CONFIG_XXX) even if the config is a boolean.
Use the correct macro for conditional compiles
Warning: This can get tricky for header files! Review the following details carefully.
If a CONFIG_XXX is set to module (=m), the build system automatically defines CONFIG_XXX_MODULE. If your driver is controlled by CONFIG_XXX and you want to check if your driver is being compiled as a module, use the following guidelines:

In the C file (or any source file that isn't a header file) for your driver, don't use #ifdef CONFIG_XXX_MODULE as it's unnecessarily restrictive and breaks if the config is renamed to CONFIG_XYZ. For any non-header source file that's compiled into a module, the build system automatically defines MODULE for the scope of that file. Therefore, to check if a C file (or any non-header source file) is being compiled as part of a module, use #ifdef MODULE (without the CONFIG_ prefix).

In header files, the same check is tricker because header files aren't compiled directly into a binary but rather compiled as part of a C file (or other source files). Use the following rules for header files:

For a header file that uses #ifdef MODULE, the result changes based on which source file is using it. This means the same header file in the same build can have different parts of its code compiled for different source files (module versus built-in or disabled). This can be useful when you want to define a macro that needs to expand one way for built-in code and expand in a different way for a module.

For a header file that needs to compile in a piece of code when a specific CONFIG_XXX is set to module (regardless of whether the source file including it is a module), the header file must use #ifdef CONFIG_XXX_MODULE.

Was this helpful?

Yes, that vendor module guide is extremely **helpful** and **essential** if you're developing kernel modules in the Android GKI (Generic Kernel Image) environment.

Here's a **concise summary** and breakdown to help you understand and apply the rules more effectively:

✅ Vendor Module Guidelines – Summary & Practical Tips

📦 1. Types of Modules

- **Library Modules**: Provide APIs (e.g., encryption, logbuffer). Not hardware-specific.
- **Driver Modules**: Bind to hardware (e.g., UART, PCIe). Must probe a device.

🧠 2. Init/Exit Best Practices

- Use `module_platform_driver()` or similar macros for drivers.
- For modules with multiple drivers:
  - Try to combine or split them.

- **Library modules** can use plain `module_init/exit()`.

🔍 3. Use `MODULE_DEVICE_TABLE()`

- Allows Android to know what devices a driver supports.
- Helps **avoid loading irrelevant drivers**.

❌ 4. Avoid CRC Mismatches (ABI Issues)

- Don't include headers just to access private struct definitions.
  - E.g., Don't include `fwnode.h` to access `struct fwnode_handle`.
- This leads to **ABI breakages and MODVERSION mismatches**.

🔒 5. Don't Touch Core Kernel Structs Directly

- Never modify or dereference:
  - `struct device`
  - `struct dev_links_info`
- Use only public APIs like:
  - `device_link_add()`
  - `devm_*()` functions

🌳 6. Device Tree (DT) Rules

- If DT node has `compatible`, **you must write a driver**.
- Don't parse such DT nodes manually via `of_find_*()`.
- Violates fw_devlink rules, which control probe order and dependency.

📎 7. Use Phandles in DT

✅ Correct:

```dts
touch-device {
    core-supply = <&ldo4>;
    sensor-supply = <&ldo10>;
};
```

❌ Bad (legacy or broken):

```dts
touch-device {
    fizz,core-regulator = "acme-pmic-ldo4";
};
```

💥 8. Don't Modify Framework API Error Codes

- Always return `-EPROBE_DEFER` when resource not ready.
- Don't convert it to another error (like `-EINVAL`).

🧹 9. Use `devm_*()` APIs

- Automatically releases resources on failure or unbind.
- Simplifies cleanup:

```c
clk = devm_clk_get(dev, NULL);
```

🔁 10. Proper Unbinding

- Implement **`remove()`** if:
  - You don't use **`devm_*()`** for everything
  - You need shutdown or cleanup

⛔ 11. Don't Load Firmware in `probe()`

- Storage may not be mounted yet.
- Defer to runtime (e.g., on first open).

⚡ 12. Use Asynchronous Probing

- Enables future optimizations (parallel loading).

```c
.driver = {
    ...
    .probe_type = PROBE_PREFER_ASYNCHRONOUS,
}
```

🧩 13. Avoid `MODULE_SOFTDEP()`

- Unreliable for controlling probe order.
- Use **`fw_devlink`** + phandles to express dependencies.

🔁 14. Use `IS_ENABLED()` Instead of `#ifdef`

✅ Correct:

```c
#if IS_ENABLED(CONFIG_XYZ)
```

❌ Wrong:

```c
#ifdef CONFIG_XYZ
```

Reason: **`IS_ENABLED()`** works for both **`=y`** and **`=m`**.

📁 15. Use `#ifdef MODULE` Carefully in Headers

- In C files: use **`#ifdef MODULE`**
- In headers:
  - For config checks: use **`CONFIG_XXX_MODULE`**
  - To check if file is used in module context: **`#ifdef MODULE`**

🏁 Final Checklist for Robust Vendor Modules

✅ Must Do	❌ Don't Do
Use **`MODULE_DEVICE_TABLE()`**	Access private structs
Use **`devm_*()`**	Modify **`struct device`** directly
Use phandles in DT	Parse DT with **`of_find_*()`**
Return **`-EPROBE_DEFER`**	Remap errors
Use **`IS_ENABLED()`**	Use **`#ifdef CONFIG_XXX`** blindly
Handle **`remove()`** or use **`devm_*()`**	Leave unbind logic undefined
Asynchronous probing support	Use **`MODULE_SOFTDEP()`**

Let me know if you want:

- A **template vendor driver**
- Real examples for **`module_platform_driver()`**, **`devm_*()`**, DT phandles
- Checklist for **GKI compliance** and testing

64. Loadable kernel modules

bookmark_border
As part of the module kernel requirements introduced in Android 8.0, all system-on-chip (SoC) kernels must support loadable kernel modules.

Kernel configuration options
To support loadable kernel modules, android-base.config in all common kernels includes the following kernel-config options (or their kernel-version equivalent):

CONFIG_MODULES=y
CONFIG_MODULE_UNLOAD=y
CONFIG_MODVERSIONS=y
All device kernels must enable these options. Kernel modules should also support unloading and reloading whenever possible.

Note: CONFIG_MODULE_SRCVERSION_ALL is optional and isn't tested against.
Module signing
Module-signing is not supported for GKI vendor modules. On devices required to support verified boot, Android requires kernel modules to be in the partitions that have dm-verity enabled. This removes the need for signing individual modules for their authenticity. Android 13 introduced the concept of GKI modules. GKI modules use the kernel's build time signing infrastructure to differentiate between GKI and other modules at run time. Unsigned modules are allowed to load as long as they only use symbols appearing on the allowlist or provided by other unsigned modules. To facilitate GKI modules signing during GKI build using kernel's build time key pair, GKI kernel config has enabled CONFIG_MODULE_SIG_ALL=y. To avoid signing non-GKI modules during device kernel builds, you must add # CONFIG_MODULE_SIG_ALL is not set as part of your kernel config fragments.

File locations
While Android 7.x and lower don't mandate against kernel modules (and include support for insmod and rmmod), Android 8.x and higher recommend the use of kernel modules in the ecosystem. The following table shows potential board-specific peripheral support required across three Android boot modes.

Boot mode Storage Display Keypad Battery PMIC Touchscreen NFC, Wi-Fi,
Bluetooth Sensors Camera
Recovery
Charger
Android
In addition to availability in Android boot modes, kernel modules may also be categorized by who owns them (the SoC vendor or the ODM). If kernel modules are being used, requirements for their placement in file system are as follows:

All kernels should have built-in support for booting and mounting partitions.
Kernel modules must be loaded from a read-only partition.
For devices required to have verified boot, kernel modules should be loaded from verified partitions.
Kernel modules shouldn't be located in /system.
GKI modules required for the device should be loaded from /system/lib/modules which is a symbolic link to /system_dlkm/lib/modules.
Kernel modules from the SoC vendor that are required for full Android or Charger modes should be located in /vendor/lib/modules.
If an ODM partition exists, kernel modules from the ODM that are required for full Android or Charger modes should be located in /odm/lib/modules. Otherwise, these modules should be located in /vendor/lib/modules.
Kernel modules from the SoC vendor and ODM that are required for Recovery mode should be located in the recovery ramfs at /lib/modules.
Kernel modules required for both Recovery mode and full Android or Charger modes should exist both in the recovery rootfs and either the /vendor or /odm partitions (as described above).
Kernel modules used in Recovery mode shouldn't depend on modules located only in /vendor or /odm, as those partitions aren't mounted in Recovery mode.
SoC vendor kernel modules shouldn't depend on ODM kernel modules.
In Android 7.x and lower, /vendor and /odm partitions are not mounted early. In Android 8.x and higher, to make module loading from these partitions possible, provisions have been made to mount partitions early for both non-A/B and A/B devices. This also ensures that the partitions are mounted in both Android and Charger modes.

Android build system support
In BoardConfig.mk, the Android build defines a BOARD_VENDOR_KERNEL_MODULES variable that provides a full list of the kernel modules intended for the vendor image. The modules listed in this variable are copied into the vendor image at /lib/modules/, and, after being mounted in Android, appear in /vendor/lib/modules (in accordance with the above requirements). Example configuration of the vendor kernel modules:

vendor_lkm_dir := device/$(vendor)/lkm-4.x

BOARD_VENDOR_KERNEL_MODULES := \
  $(vendor_lkm_dir)/vendor_module_a.ko \
  $(vendor_lkm_dir)/vendor_module_b.ko \
  $(vendor_lkm_dir)/vendor_module_c.ko
In this example, a vendor kernel module pre-built repository is mapped into the Android build at the location listed above.

The recovery image may contain a subset of the vendor modules. The Android build defines the variable BOARD_RECOVERY_KERNEL_MODULES for these modules. Example:

vendor_lkm_dir := device/$(vendor)/lkm-4.x
BOARD_RECOVERY_KERNEL_MODULES := \
  $(vendor_lkm_dir)/vendor_module_a.ko \
  $(vendor_lkm_dir)/vendor_module_b.ko
The Android build takes care of running depmod to generate the required modules.dep files in /vendor/lib/modules and /lib/modules (recovery ramfs).

Module loading and versioning
Load all kernel modules in one pass from init.rc* by invoking modprobe -a. This avoids the overhead of repeatedly initializing the C runtime environment for the modprobe binary. The early-init event can be modified to invoke modprobe:

on early-init
    exec u:r:vendor_modprobe:s0 -- /vendor/bin/modprobe -a -d \
        /vendor/lib/modules module_a module_b module_c ...
Typically, a kernel module must be compiled with the kernel that the module is to be used with (otherwise the kernel refuses to load the module). CONFIG_MODVERSIONS provides a workaround by detecting breakages in the application binary interface (ABI). This feature calculates a cyclic redundancy check (CRC) value for the prototype of each exported symbol in the kernel and stores the values as part of the kernel; for symbols used by a kernel module, the values are also stored in the kernel module. When the module is loaded, the values for the symbols used by the module are compared with the ones in the kernel. If the values match, the module is loaded; otherwise the load fails.

Note: For help comparing, tracking, and mitigating kernel ABI changes that affect compatibility with kernel modules, refer to the scripts and utilities in ABI Monitoring Utilities.
To enable the updating of the kernel image separately from the vendor image, enable CONFIG_MODVERSIONS. Doing so allows small updates to the kernel (such as bug fixes from LTS) to be made while maintaining compatibility with existing kernel modules in the vendor image. However, CONFIG_MODVERSIONS doesn't fix an ABI breakage by itself. If the prototype of an exported symbol in the kernel changes, either due to modification of the source or because the kernel configuration changed, this breaks compatibility with kernel modules that use that symbol. In such cases, the kernel module must be recompiled.

For example, the task_struct structure in the kernel (defined in include/linux/sched.h) contains many fields conditionally included depending on the kernel configuration. The sched_info field is present only if CONFIG_SCHED_INFO is enabled (which occurs when CONFIG_SCHEDSTATS or CONFIG_TASK_DELAY_ACCT are enabled). If these configuration options change state, the layout of the task_struct structure changes and any exported interfaces from the kernel that use task_struct are altered (for example, set_cpus_allowed_ptr in kernel/sched/core.c). Compatibility with previously compiled kernel modules that use these interfaces breaks, requiring those modules to be rebuilt with the new kernel configuration.

For more details on CONFIG_MODVERSIONS, refer to the documentation in the kernel tree at Documentation/kbuild/modules.rst.

Was this helpful?

Yes, the information you've provided about **Loadable Kernel Modules (LKMs)** in Android is **very helpful**—especially for understanding Android's requirements and best practices for supporting and using vendor modules across different boot modes and partitioning schemes.

Here's a **condensed and practical summary** you can use as a reference:

🔧 Loadable Kernel Modules in Android – Key Points

🧩 1. Kernel Config Requirements

In **`android-base.config`**:

```text
CONFIG_MODULES=y
CONFIG_MODULE_UNLOAD=y
```

```
CONFIG_MODVERSIONS=y
```

- These enable basic module support, unloading, and ABI compatibility checking.
- **Optional**: `CONFIG_MODULE_SRCVERSION_ALL`
- **GKI builds only**:
  - `CONFIG_MODULE_SIG_ALL=y` → signs only **GKI** modules
  - For vendor builds: `# CONFIG_MODULE_SIG_ALL is not set`

📦 2. File Placement Rules

Partition	Purpose	Module Location (on device)
/vendor	SoC vendor modules (Android + Charger mode)	`/vendor/lib/modules/`
/odm	ODM modules (if ODM partition exists)	`/odm/lib/modules/`
/recovery	Recovery mode modules	`/lib/modules/` (in ramfs)
/system_dlkm	GKI modules (symlinked from `/system/lib/modules`)	`/system_dlkm/lib/modules/`

> 📌 Modules used in both Android and Recovery must exist in both locations.
> ❌ **Do not put modules in `/system`.**

🏗️ 3. Android Build Integration

📁 In `BoardConfig.mk`:

```make
# Vendor kernel modules
BOARD_VENDOR_KERNEL_MODULES := \
  device/vendor/lkm-4.x/vendor_module_a.ko \
  device/vendor/lkm-4.x/vendor_module_b.ko

# Recovery-specific modules
BOARD_RECOVERY_KERNEL_MODULES := \
  device/vendor/lkm-4.x/vendor_module_a.ko
```

- Android build system copies them into `/vendor/lib/modules/` and `/lib/modules/` respectively.
- **Build automatically runs `depmod`** to generate `modules.dep`.

🚀 4. Module Loading (modprobe)

Use a single `modprobe -a` in `init.rc` to load all needed modules:

```rc
on early-init
    exec u:r:vendor_modprobe:s0 -- /vendor/bin/modprobe -a -d \
        /vendor/lib/modules module_a module_b module_c
```

> ✅ Avoid repeated `modprobe` calls (saves C-runtime startup overhead).

🔐 5. Module Signing in GKI Context

- **GKI modules** are signed with a GKI key (`CONFIG_MODULE_SIG_ALL=y`)
- **Vendor modules** are **not signed**, but:
  - Must come from **dm-verity protected partitions**
  - Can use **allowlisted symbols** or other unsigned symbols
  - Not allowed to override **protected GKI modules** unless they're explicitly made unprotected

---

🧬 6. ABI Compatibility with `CONFIG_MODVERSIONS`

- Adds CRC to symbol prototypes
- Ensures **compatibility of vendor modules** across kernel updates
- ❗ **Does NOT fix ABI breaks**—it just detects them
  - If a struct's layout changes (e.g., `task_struct`), modules must be **rebuilt**

🔁 Example ABI Break Case

If kernel config changes:

```c
task_struct {
  ...
  #ifdef CONFIG_SCHED_INFO
    struct sched_info sched_info;
  #endif
}
```

→ `set_cpus_allowed_ptr()` uses `task_struct`
→ CRC of symbol changes
→ Modules compiled with old layout will **fail to load**.

🧪 7. Tools to Manage ABI

- Use the ABI monitoring utilities:

  ```
  abi-compliance-checker
  abi-dumper
  abi-monitoring-tools
  ```

They help track `.stg` files, KMI symbol list, `.crc` mismatches, etc.

✅ Summary Checklist

Task	Status
Enable required kernel configs	✅
Avoid module signing for vendor builds	✅
Use correct partition for each module	✅
Declare modules in `BoardConfig.mk`	✅
Use `modprobe -a` for batch loading	✅
Enable `CONFIG_MODVERSIONS`	✅
Rebuild vendor modules on ABI change	⚠️
Avoid using GKI protected symbols	✅

Let me know if you want:

- A minimal example vendor module with Makefile, DT support, and `.ko` integration
- A `modprobe` init.rc snippet generator
- ABI checker setup walkthrough

65. ↪
`modprobe`
what is this

`modprobe` is a **Linux command-line tool** used to **load or unload kernel modules** — especially when the modules have **dependencies**.

🔍 What is `modprobe`?

`modprobe`:

- Loads a kernel module (`.ko` file) **into the running kernel**.
- Automatically loads any **dependent modules** listed in `modules.dep`.
- Is safer and smarter than using `insmod` or `rmmod` directly.

✅ Example Usage

Load a single module:

```bash
modprobe my_module
```

Load multiple modules at once:

```bash
modprobe -a module1 module2 module3
```

Unload a module:

```bash
modprobe -r my_module
```

Use a custom module directory (common in Android):

```bash
modprobe -d /vendor/lib/modules my_module
```

🔗 How is `modprobe` different from `insmod`?

Command	Description
`insmod`	Loads **only one** module, **no dependencies**
`modprobe`	Loads the module **and its dependencies**

📦 Where does `modprobe` get dependency info?

From:

```text
/lib/modules/<kernel-version>/modules.dep
```

> This file is created by the `depmod` tool during build.

📱 In Android?

Android’s **`init.rc`** uses **`modprobe`** to load vendor modules early in the boot process:

```rc
on early-init
  exec u:r:vendor_modprobe:s0 -- /vendor/bin/modprobe -a -d /vendor/lib/modules my_driver
```

This:

- Loads **`my_driver.ko`**
- From the **`/vendor/lib/modules`** folder
- With all its dependencies (if any)

Let me know if you want:

- A minimal working example using **`modprobe`** on Android
- How **`modprobe`** uses **`modules.dep`**
- How to configure and generate it in your kernel build

66. Kernel module support

bookmark_border
A generic kernel image (GKI) may not contain the required driver support to enable a device to mount partitions. To enable a device to mount partitions and to continue booting, first-stage init is enhanced to load the kernel modules present on a ramdisk. The ramdisk is split into generic and vendor ramdisks. Vendor kernel modules are stored in the vendor ramdisk. The order in which kernel modules are loaded is configurable.

Module location
The ramdisk is the filesystem for first-stage init, and for the recovery/fastbootd image on A/B and virtual A/B devices. It's an initramfs composed of two cpio archives that get concatenated by the bootloader. The first cpio archive, which is stored as the vendor ramdisk in the vendor-boot partition, contains these components:

First-stage init vendor kernel modules, located in /lib/modules/.
modprobe config files, located in /lib/modules/: modules.dep, modules.softdep, modules.alias, modules.options.
A modules.load file that indicates which modules to load during first stage init, and in which order, in /lib/modules/.
Vendor recovery-kernel modules, for A/B and Virtual A/B devices, in /lib/modules/
modules.load.recovery which indicates the modules to load, and in which order, for A/B and Virtual A/B devices, in /lib/modules.
The second cpio archive, which is supplied with the GKI as the ramdisk of the boot.img and applied on top of the first, contains first_stage_init and the libraries on which it depends.

Module loading in first-stage init
First-stage init begins by reading the modprobe configuration files from /lib/modules/ on the ramdisk. Next, it reads the list of modules specified in /lib/modules/modules.load (or in the case of recovery, /lib/modules/modules.load.recovery) and attempts to load each of those modules in order, following the configuration specified in the previously loaded files. The requested order may be deviated from to satisfy hard or soft dependencies.

Build support, first-stage init
To specify kernel modules to be copied into the vendor ramdisk cpio, list them in BOARD_VENDOR_RAMDISK_KERNEL_MODULES. The build runs depmod on these modules and puts the resulting modprobe configuration files in the vendor ramdisk cpio.

The build also creates a modules.load file and stores it in the vendor ramdisk cpio. By default it contains all of the modules listed in BOARD_VENDOR_RAMDISK_KERNEL_MODULES. To override the contents of that file, use BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD, as shown in this example:

```
BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD := \
    device/vendor/mydevice-kernel/first.ko \
    device/vendor/mydevice-kernel/second.ko \
    device/vendor/mydevice-kernel/third.ko
```

Build support, full Android
As is the case in Android 10 and lower releases, kernel modules listed in BOARD_VENDOR_KERNEL_MODULES are copied by the Android platform build into the vendor partition at /vendor/lib/modules. The platform build runs depmod on these modules, and copies the depmod output files into the vendor partition at the same location. The mechanism for loading kernel modules from /vendor remains the same as it was for prior releases of Android. It's your decision how and when to load these modules, although typically this is done using init.rc scripts.

Wildcards and integrated kernel builds
Vendors who combine their device kernel build with the Android platform build may run into a problem using the above mentioned BOARD macros to specify kernel modules to be copied on to the device. If the vendor wishes to avoid listing kernel modules in the device's platform build files, they can use a wildcard ($(wildcard device/vendor/mydevice/*.ko). Note that the wildcard doesn't work in the case of an integrated kernel build, because when make is invoked and the macros are expanded in makefiles, the kernel modules haven't been built, so the macros are empty.

To get around this problem, the vendor may have their kernel build create a zip archive containing the kernel modules to be be copied onto each partition. Set the path of that zip archive in BOARD_*_KERNEL_MODULES_ARCHIVE where * is the name of the partition (such as BOARD_VENDOR_KERNEL_MODULES_ARCHIVE). The Android platform build extracts this zip archive into the appropriate location and runs depmod on the modules.

The kernel module zip archive should have a make rule that ensures the platform build can generate the archive when required.

Recovery
In prior Android releases, kernel modules required for recovery were specified in BOARD_RECOVERY_KERNEL_MODULES. In Android 12, kernel modules required for recovery are still specified using this macro. However, the recovery kernel modules are copied to the vendor ramdisk cpio,

rather than the generic ramdisk cpio. By default all kernel modules listed in BOARD_RECOVERY_KERNEL_MODULES are loaded during first-stage init. If you only want a subset of these modules to be loaded, specify the contents of that subset in BOARD_RECOVERY_KERNEL_MODULES_LOAD.

This documentation explains **how Android handles loading kernel modules** — especially in early boot (first-stage init), full Android, and recovery — under the **GKI (Generic Kernel Image)** model.

Here's a simplified breakdown:

🔧 What Is This About?

Android needs to load kernel modules (drivers) during boot, but with **GKI**, many drivers are not built-in. So Android:

- Loads vendor-specific kernel modules from the **vendor ramdisk** early in boot.
- Loads other modules (for full Android mode) from **/vendor/lib/modules** or **/odm/lib/modules**.

📂 Where Kernel Modules Are Stored

Location	Purpose
`/lib/modules/` on **vendor ramdisk**	Modules for **first-stage init** and **recovery**
`/vendor/lib/modules/`	Vendor modules for full Android mode
`/odm/lib/modules/`	ODM-specific modules (if ODM partition exists)
`/system/lib/modules/` → symlink to `/system_dlkm/lib/modules`	GKI-provided modules

🚀 How Modules Are Loaded During Boot

✅ First-Stage Init (early boot):

- `first_stage_init` reads:
  - `/lib/modules/modules.dep` and other configs (from `depmod`)
  - `/lib/modules/modules.load` — list of `.ko` files to load
- Loads each module in that list using `modprobe`, resolving dependencies.

✅ Recovery Mode:

- Uses `/lib/modules/modules.load.recovery` instead.

⚙️ How to Configure Module Loading

1. For First-Stage Init

```
make

BOARD_VENDOR_RAMDISK_KERNEL_MODULES := \
    path/to/first.ko \
    path/to/second.ko
```

If you want to specify **load order**:

```
make

BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD := \
    path/to/first.ko \
    path/to/second.ko
```

2. For Recovery Mode

```
make

BOARD_RECOVERY_KERNEL_MODULES := \
    path/to/recovery_driver.ko
```

```
BOARD_RECOVERY_KERNEL_MODULES_LOAD := \
    path/to/recovery_driver.ko
```

3. For Full Android

```make
BOARD_VENDOR_KERNEL_MODULES := \
    path/to/main_driver.ko
```

🛠️ For Integrated Kernel Builds (when `*.ko` is built as part of kernel)

Wildcards like **`$(wildcard path/*.ko)`** won't work because `.ko` files don't exist yet.

✅ Workaround:

Build a ZIP file from your kernel build:

```make
BOARD_VENDOR_KERNEL_MODULES_ARCHIVE := path/to/vendor_modules.zip
```

> The build system will extract it, run **`depmod`**, and place everything properly.

📌 Summary

What You Want	What to Use
Load vendor modules early in boot	**`BOARD_VENDOR_RAMDISK_KERNEL_MODULES`**
Load vendor modules only in recovery	**`BOARD_RECOVERY_KERNEL_MODULES`**
Load modules during full Android boot	**`BOARD_VENDOR_KERNEL_MODULES`**
Specify module load order	Use the **`*_LOAD`** variants
Integrated kernel build workaround	Use **`BOARD_*_KERNEL_MODULES_ARCHIVE`** zip

Would you like an actual example **`BoardConfig.mk`** showing all of these?

67. what is full android here

In the context of Android kernel module support and boot flow, **"Full Android"** refers to the **normal Android operating mode**, when the device has booted all the way to the regular Android user interface (Home screen, apps, etc.).

✅ Full Android Mode

This is the **default, normal boot mode** where:

- The full Android system (`/system`, `/vendor`, `/odm`, etc.) is mounted.
- All services and apps are running.
- User interacts with the UI.

In this mode:

- Kernel modules required for **Wi-Fi, Bluetooth, GPU, audio, camera, sensors, etc.** are usually loaded from `/vendor/lib/modules` or `/odm/lib/modules`.
- These modules are **not needed** during first-stage init or recovery, so they are not included in the **vendor ramdisk**.
- Modules may be loaded by `init.rc` **scripts** or **udev-like mechanisms** (e.g., via `modprobe`).

🔁 Comparison with Other Boot Modes

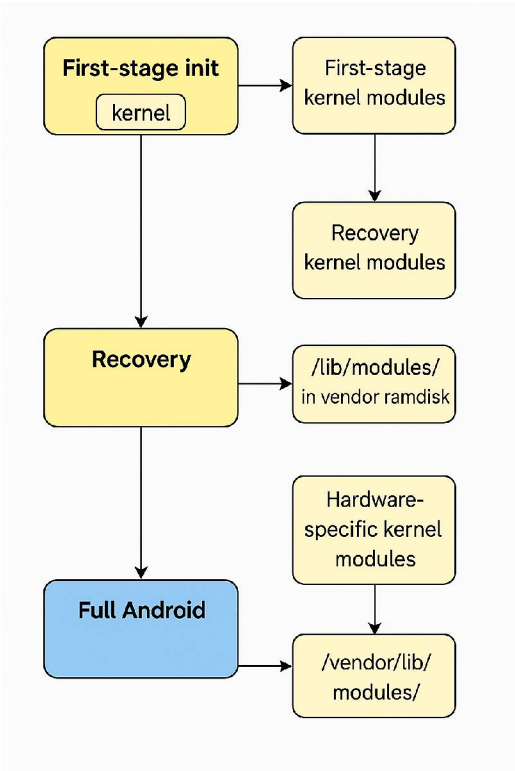
Boot Mode	Purpose	Mounts	Kernel Modules Loaded From
First-stage init	Minimal environment for mounting partitions	ramdisk only	`/lib/modules/` in **vendor ramdisk**
Recovery	For updates or factory reset	ramdisk only	`/lib/modules/` in **vendor ramdisk**
Charger Mode	While charging without full system boot	`/vendor` may be mounted	`/vendor/lib/modules/`
Full Android	Normal OS mode for user interaction	Full Android FS (`/system`, `/vendor`, etc.)	`/vendor/lib/modules/`, `/odm/lib/modules/`

📦 Key Takeaways

- **"Full Android"** means Android is completely booted and running.
- Modules for hardware like display, camera, NFC, etc. are loaded in this mode.
- These modules are placed in the vendor or odm partitions.
- They are **not required** in recovery or early boot unless that hardware must function during those modes.

Would you like a boot sequence diagram showing when and where each type of module is loaded (first-stage, recovery, full Android)?

68. ↪
Would you like an actual example `BoardConfig.mk` showing all of these?
yes



69. Test GKI modules

bookmark_border
Test your GKI kernel and modules using the system_dlkm.img image and the system_dlkm dynamic partition.

Before you can test using the development-only system_dlkm.img image built from GKI kernel repository, you must disable the dm-verity security feature to boot from the image. For further information on dm-verity, refer to Implement dm-verity.

The system_dlkm.img image from kernel release artifacts is only for development use and cannot be used to run CTS/VTS tests or use in the production. To run CTS/VTS tests, you need to use the production images and signed modules from the system_dlkm_staging_archive.tar.gz archive in the kernel artifacts. You can build the system_dlkm partition image as described in Implement a GKI module partition.

In this context, **testing GKI modules** involves verifying that your **Generic Kernel Image (GKI)** and its associated **loadable modules** function correctly within the Android system. Here's a summary of what's going on:

🔧 How to Test GKI Modules

There are **two key phases** depending on whether you are in **development** or **preparing for production**:

✅ 1. Testing with `system_dlkm.img` (Development Only)

- `system_dlkm.img` is generated from the GKI kernel repo.
- It contains GKI modules meant for **development testing only**.
- It **cannot be used for CTS/VTS or production**.

🔐 Disable dm-verity

To boot with this image, you **must disable `dm-verity`** (Android's filesystem integrity check), because:

- The modules in this image are **not signed**.
- Disabling dm-verity allows you to boot an image not protected by verified boot.

📘 See: Implement dm-verity

🔒 2. Testing for Production (CTS/VTS Compatible)

To pass certification and run compliance tests:

- Use `system_dlkm_staging_archive.tar.gz` from GKI **kernel artifacts**.
- This archive contains:
  - Properly signed GKI `.ko` modules.
  - A reproducible and ABI-compliant set of modules for production use.
- You must use this archive to **build your own `system_dlkm.img`** for production.
  - Refer to: Implement a GKI module partition

📦 About `system_dlkm`

- A **dynamic partition** introduced to store **kernel modules** separately.
- Stands for **Dynamic Loadable Kernel Modules** partition.
- Located at: `/system_dlkm/lib/modules`

✅ Summary

Use Case	Image/Archive	dm-verity	CTS/VTS	Notes
Dev Testing	**`system_dlkm.img`**	Disabled	❌	From GKI repo
Production Testing / CTS	**`system_dlkm_staging_archive.tar.gz`** → build system_dlkm.img	Enabled	✅	Must use signed modules

Let me know if you want a diagram or a step-by-step test workflow!

70. Boot time optimization

bookmark_border
This page provides tips to improve boot time.

Strip debug symbols from modules
Similar to how debug symbols are stripped from the kernel on a production device, make sure you also strip the debug symbols from modules. Stripping debug symbols from modules helps boot time by reducing the following:

The time it takes to read the binaries from flash.
The time it takes to decompress the ramdisk.
The time it takes to load the modules.
Stripping debug symbol from modules may save several seconds during boot.

Symbol stripping is enabled by default in the Android platform build, but to explicitly enable them, set BOARD_DO_NOT_STRIP_VENDOR_RAMDISK_MODULES in your device-specific config under device/vendor/device.

Use LZ4 compression for kernel and ramdisk
Gzip generates a smaller compressed output compared to LZ4, but LZ4 decompresses faster than Gzip. For the kernel and modules, the absolute storage size reduction from using Gzip isn't that significant compared to the decompression time benefit of LZ4.

Support for LZ4 ramdisk compression has been added to the Android platform build through BOARD_RAMDISK_USE_LZ4. You can set this option in your device-specific config. Kernel compression can be set through kernel defconfig.

Switching to LZ4 should give 500ms to 1000ms faster boot time.

Avoid excessive logging in your drivers
In ARM64 and ARM32, function calls that are more than a specific distance from the call site need a jump table (called a procedure linking table, or PLT) to be able to encode the full jump address. Since modules are loaded dynamically, these jump tables need to be fixed up during module load. The calls that need relocation are called relocation entries with explicit addends (or RELA, for short) entries in the ELF format.

The Linux kernel does some memory size optimization (such as cache hit optimization) when allocating the PLT. With this upstream commit, the optimization scheme has an O(N^2) complexity, where N is the number of RELAs of type R_AARCH64_JUMP26 or R_AARCH64_CALL26. So having fewer RELAs of these types is helpful in reducing the module load time.

One common coding pattern that increases the number of R_AARCH64_CALL26 or R_AARCH64_JUMP26 RELAs is excessive logging in a driver. Each call to printk() or any other logging scheme typically adds a CALL26/JUMP26 RELA entry. In the commit text in the upstream commit, ,notice that even with the optimization, the six modules take about 250ms to load—that is because those six modules were the top six modules with the most amount of logging.

Reducing logging can save can save about 100 - 300ms on boot times depending on how excessive the existing logging is.

Enable asynchronous probing, selectively
When a module is loaded, if the device that it supports has already been populated from the DT (devicetree) and added to driver core, then the device probe is done in the context of the module_init() call. When a device probe is done in the context of module_init(), the module can't finish loading until the probe completes. Since module loading is mostly serialized, a device that takes a relatively long time to probe slows the boot time.

To avoid slower boot times, enable asynchronous probing for modules that take a while to probe their devices. Enabling asynchronous probing for all modules might not be beneficial as the time it takes to fork a thread and kick off the probe might be as high as the time it takes to probe the device.

Devices that are connected through a slow bus such as I2C, devices that do firmware loading in their probe function, and devices that do a lot of hardware initialization can lead to the timing issue. The best way to identify when this happens is to collect the probe time for every driver and sort it.

To enable asynchronous probing for a module, it isn't sufficient to only set the PROBE_PREFER_ASYNCHRONOUS flag in the driver code. For modules, you also need to add module_name.async_probe=1 in the kernel command line or pass async_probe=1 as a module parameter when loading the module using modprobe or insmod.

Enabling asynchronous probing can save about 100 - 500ms on boot times depending on your hardware/drivers.

Probe your CPUfreq driver as early as possible

The earlier your CPUfreq driver probes, the sooner you can scale the CPU frequency to maximum (or some thermally limited maximum) during boot. The faster the CPU, the faster the boot. This guideline also applies to devfreq drivers that control the DRAM, memory, and interconnect frequency.

With modules, the load ordering can depend on the initcall level and compile or link order of the drivers. Use an alias MODULE_SOFTDEP() to make sure the cpufreq driver is among the first few modules to load.

Apart from loading the module early, you also need to make sure all the dependencies to probe the CPUfreq driver have also probed. For example, if you need a clock or regulator handle to control the frequency of your CPU, make sure they are probed first. Or you might need thermal drivers to be loaded before the CPUfreq driver if it is possible for your CPUs to get too hot during boot up. So, do what you can to make sure the CPUfreq and relevant devfreq drivers probe as early as possible.

The savings from probing your CPUfreq driver early can be very small to very large depending on how early you can get these to probe and at what frequency the bootloader leaves the CPUs in.

Move modules to second stage init, vendor or vendor_dlkm partition
Because the first stage init process is serialized, there aren't many opportunities to parallelize the boot process. If a module isn't needed for first stage init to finish, move the module to second stage init by placing it in the vendor or vendor_dlkm partition.

First stage init doesn't require probing several devices to get to second stage init. Only console and flash storage capabilities are needed for a normal boot flow.

Load the following essential drivers:

watchdog
reset
cpufreq
For recovery and user space fastbootd mode, first stage init requires more devices to probe (such as USB), and display. Keep a copy of these modules in the first stage ramdisk and in the vendor or vendor_dlkm partition. This lets them be loaded in first stage init for recovery or fastbootd boot flow. However, don't load the recovery mode modules in first stage init during normal boot flow. Recovery mode modules can be deferred to second stage init to decrease the boot time. All other modules that aren't needed in first stage init should be moved to the vendor or vendor_dlkm partition.

Given a list of leaf devices (for example, the UFS or serial), dev needs.sh script finds all drivers, devices, and modules needed for dependencies or suppliers (for example, clocks, regulators, or gpio) to probe.

Moving modules to second stage init decreases boot times in the following ways:

Ramdisk size reduction.
This yields faster flash reads when the bootloader loads the ramdisk (serialized boot step).
This yields faster decompression speeds when the kernel decompresses the ramdisk (serialized boot step).
Second stage init works in parallel, which hides the module's loading time with the work being done in second stage init.
Moving modules to second stage can save 500 - 1000ms on boot times depending on how many modules you're able to move to second stage init.

Module loading logistics
The latest Android build features board configurations that control which modules copy over to each stage, and which modules load. This section focuses on the following subset:

BOARD_VENDOR_RAMDISK_KERNEL_MODULES. This list of modules to be copied into the ramdisk.
BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD. This list of modules to be loaded in first stage init.
BOARD_VENDOR_RAMDISK_RECOVERY_KERNEL_MODULES_LOAD. This list of modules to be loaded when recovery or fastbootd is selected from the ramdisk.
BOARD_VENDOR_KERNEL_MODULES. This list of modules to be copied into the vendor or vendor_dlkm partition at /vendor/lib/modules/ directory.
BOARD_VENDOR_KERNEL_MODULES_LOAD. This list of modules to be loaded in second stage init.
The boot and recovery modules in ramdisk must also be copied to the vendor or vendor_dlkm partition at /vendor/lib/modules. Copying these modules to the vendor partition ensures the modules aren't invisible during second stage init, which is useful for debugging and collecting modinfo for bugreports.

The duplication should cost minimal space on the vendor or vendor_dlkm partition as long as the boot module set is minimized. Make sure that the vendor's modules.list file has a filtered list of modules in /vendor/lib/modules. The filtered list ensures boot times aren't affected by the modules loading again (which is an expensive process).

Ensure that recovery mode modules load as a group. Loading recovery mode modules can be done either in recovery mode, or at the beginning of the second stage init in each boot flow.

You can use the device Board.Config.mk files to perform these actions as seen in the following example:

```
# All kernel modules
KERNEL_MODULES := $(wildcard $(KERNEL_MODULE_DIR)/*.ko)
KERNEL_MODULES_LOAD := $(strip $(shell cat $(KERNEL_MODULE_DIR)/modules.load)

# First stage ramdisk modules
BOOT_KERNEL_MODULES_FILTER := $(foreach m,$(BOOT_KERNEL_MODULES),%/$(m))

# Recovery ramdisk modules
RECOVERY_KERNEL_MODULES_FILTER := $(foreach m,$(RECOVERY_KERNEL_MODULES),%/$(m))
BOARD_VENDOR_RAMDISK_KERNEL_MODULES += \
    $(filter $(BOOT_KERNEL_MODULES_FILTER) \
            $(RECOVERY_KERNEL_MODULES_FILTER),$(KERNEL_MODULES))

# ALL modules land in /vendor/lib/modules so they could be rmmod/insmod'd,
# and modules.list actually limits us to the ones we intend to load.
BOARD_VENDOR_KERNEL_MODULES := $(KERNEL_MODULES)
# To limit /vendor/lib/modules to just the ones loaded, use:
# BOARD_VENDOR_KERNEL_MODULES := $(filter-out \
#     $(BOOT_KERNEL_MODULES_FILTER),$(KERNEL_MODULES))

# Group set of /vendor/lib/modules loading order to recovery modules first,
# then remainder, subtracting both recovery and boot modules which are loaded
# already.
BOARD_VENDOR_KERNEL_MODULES_LOAD := \
      $(filter-out $(BOOT_KERNEL_MODULES_FILTER), \
      $(filter $(RECOVERY_KERNEL_MODULES_FILTER),$(KERNEL_MODULES_LOAD)))
BOARD_VENDOR_KERNEL_MODULES_LOAD += \
      $(filter-out $(BOOT_KERNEL_MODULES_FILTER) \
         $(RECOVERY_KERNEL_MODULES_FILTER),$(KERNEL_MODULES_LOAD))

# NB: Load order governed by modules.load and not by $(BOOT_KERNEL_MODULES)
BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD := \
      $(filter $(BOOT_KERNEL_MODULES_FILTER),$(KERNEL_MODULES_LOAD))

# Group set of /vendor/lib/modules loading order to boot modules first,
# then the remainder of recovery modules.
BOARD_VENDOR_RAMDISK_RECOVERY_KERNEL_MODULES_LOAD := \
   $(filter $(BOOT_KERNEL_MODULES_FILTER),$(KERNEL_MODULES_LOAD))
BOARD_VENDOR_RAMDISK_RECOVERY_KERNEL_MODULES_LOAD += \
   $(filter-out $(BOOT_KERNEL_MODULES_FILTER), \
   $(filter $(RECOVERY_KERNEL_MODULES_FILTER),$(KERNEL_MODULES_LOAD)))
```

This example showcases an easier-to-manage subset of BOOT_KERNEL_MODULES and RECOVERY_KERNEL_MODULES to be specified locally in the board configuration files. The preceding script finds and fills each of the subset modules from the selected available kernel modules, leaving the reamining modules for second stage init.

For second stage init, we recommend running the module loading as a service so it doesn't block boot flow. Use a shell script to manage the module loading so that other logistics, such as error handling and mitigation, or module load completion, can be reported back (or ignored) if necessary.

You can ignore a debug module load failure that isn't present on user builds. To ignore this failure, set the vendor.device.modules.ready property to trigger later stages of init rc scripting bootflow to continue onto the launch screen. Reference the following example script, if you have the following code in /vendor/etc/init.insmod.sh:

```
#!/vendor/bin/sh
. . .
if [ $# -eq 1 ]; then
  cfg_file=$1
else
  # Set property even if there is no insmod config
  # to unblock early-boot trigger
  setprop vendor.common.modules.ready
```

```
  setprop vendor.device.modules.ready
  exit 1
fi

if [ -f $cfg_file ]; then
  while IFS="|" read -r action arg
  do
    case $action in
      "insmod") insmod $arg ;;
      "setprop") setprop $arg 1 ;;
      "enable") echo 1 > $arg ;;
      "modprobe") modprobe -a -d /vendor/lib/modules $arg ;;
     . . .
    esac
  done < $cfg_file
fi
```

In the hardware rc file, the one shot service could be specified with:

```
service insmod-sh /vendor/etc/init.insmod.sh /vendor/etc/init.insmod.<hw>.cfg
    class main
    user root
    group root system
    Disabled
    oneshot
```

Additional optimizations can be made after modules move from the first to second stage. You can use the modprobe blocklist feature to split up the second stage boot flow to include deferred module loading of nonessential modules. Loading of modules used exclusively by a specific HAL can be deferred to load the modules only when the HAL is started.

To improve apparent boot times, you can specifically choose modules in the module loading service that are more conducive to loading after the launch screen. For example, you can explicitly late load the modules for video decoder or Wi-Fi after the init boot flow has been cleared (sys.boot_complete Android property signal, for example). Make sure the HALs for the late loading modules block long enough when the kernel drivers aren't present.

Alternatively, you can use init's wait<file>[<timeout>] command in the boot flow rc scripting to wait for select sysfs entries to show that driver modules have completed the probe operations. An example of this is waiting for the display driver to complete loading in the background of recovery or fastbootd, before presenting menu graphics.

Initialize the CPU frequency to a reasonable value in the bootloader
Not all SoCs/products might be able to boot the CPU at the highest frequency due to thermal or power concerns during boot loop tests. However, make sure the bootloader sets the frequency of all the online CPUs to as high as safely possible for a SoC or product. This is very important because, with a fully modular kernel, the init ramdisk decompression takes place before the CPUfreq driver can be loaded. So, if the CPU is left at the lower end of its frequency by the bootloader, the ramdisk decompression time can take longer than a statically compiled kernel (after adjusting for ramdisk size difference) because the CPU frequency would be very low when doing CPU intensive work (decompression). The same applies to memory and interconnect frequency.

Initialize CPU frequency of big CPUs in the bootloader
Before the CPUfreq driver is loaded, the kernel is unaware of the CPU frequencies and doesn't scale the CPU sched capacity for their current frequency. The kernel might migrate threads to the big CPU if the load is sufficiently high on the little CPU.

Make sure the big CPUs are at least as performant as the little CPUs for the frequency at which the bootloader leaves them in. For example, if the big CPU is 2x as performant as the little CPU for the same frequency, but the bootloader sets the little CPU's frequency to 1.5 GHz and the big CPU's frequency to 300 MHz, then the boot performance is going to drop if the kernel moves a thread to the big CPU. In this example, if it is safe to boot the big CPU at 750 MHz, you should do so even if you don't plan to explicitly use it.

Drivers shouldn't load firmware in first stage init
There might be some unavoidable cases where firmware needs to be loaded in first stage init. But in general, drivers shouldn't load any firmware in first stage init, especially in device probe context. Loading firmware in first stage init causes the entire boot process to stall if the firmware isn't available in the first stage ramdisk. And even if the firmware is present in the first stage ramdisk, it still causes an unnecessary delay.

Here's a **simplified and structured summary** of the **boot time optimization techniques** from your detailed note, suitable for engineers tuning Android boot performance:

✅ Key Areas to Optimize Boot Time

1. Reduce Module Load Time

🔹 Strip Debug Symbols

- Removes unnecessary data from `.ko` files.
- Benefits:
  - Faster module load time
  - Smaller ramdisk → faster decompression
- Set:

```make
BOARD_DO_NOT_STRIP_VENDOR_RAMDISK_MODULES := false
```

2. Use Faster Compression

🔹 Use `LZ4` instead of `Gzip`

- LZ4 decompresses much faster.
- Set in **BoardConfig.mk**:

```make
BOARD_RAMDISK_USE_LZ4 := true
```

- Kernel:
  `CONFIG_KERNEL_COMPRESSION_LZ4=y`

⏱️ Saves **500–1000ms**.

3. Reduce Logging in Kernel Modules

- Each `printk()` call causes a relocation entry (`CALL26`).
- Excessive logging → more RELA entries → $O(N^2)$ time during module load.
- Tip: Reduce logging in production modules.

⏱️ Saves **100–300ms**.

4. Enable Asynchronous Probing

🔹 For slow probing devices (e.g., I2C, firmware load, display)

- Prevents blocking module load pipeline.
- Add to kernel cmdline:

```bash
<module>.async_probe=1
```

⏱️ Saves **100–500ms** per module.

5. Early Probing of CPUfreq/Devfreq Drivers

- Faster CPU scaling → faster decompression, init
- Use:

```c
MODULE_SOFTDEP("pre: regulator-x clock-y");
```

- Ensure clocks, thermal, regulators are probed first.

⏱️ Potentially **significant improvement** if CPU is bottleneck.

---

6. Move Unused Modules to Second Stage

- First stage init = serialized → keep it minimal
- Move drivers not needed early to:
  - `/vendor/lib/modules`
  - `/vendor_dlkm/lib/modules`

⏱️ Saves **500–1000ms**

---

7. Optimize Module Loading

Define these in `BoardConfig.mk`:

```make
BOARD_VENDOR_RAMDISK_KERNEL_MODULES :=
BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD :=
BOARD_VENDOR_KERNEL_MODULES :=
BOARD_VENDOR_KERNEL_MODULES_LOAD :=
```

Example module management script:

```sh
#!/vendor/bin/sh
if [ $# -eq 1 ]; then
  cfg_file=$1
else
  setprop vendor.device.modules.ready
  exit 1
fi

if [ -f $cfg_file ]; then
  while IFS="|" read -r action arg; do
    case $action in
      "insmod") insmod $arg ;;
      "modprobe") modprobe -a -d /vendor/lib/modules $arg ;;
      "setprop") setprop $arg 1 ;;
    esac
  done < $cfg_file
fi
```

Define service in `init.rc`:

```rc
service insmod-sh /vendor/etc/init.insmod.sh /vendor/etc/init.insmod.cfg
    class main
    user root
    group root system
    disabled
    oneshot
```

---

8. Late Load HAL-specific Modules

- E.g., Wi-Fi, Video decode → load after `sys.boot_completed`.
- Improves perceived boot time (faster to launch screen).

---

9. Init Wait for Select Drivers

- Use:

```rc
wait /sys/class/drm/card0 5
```

- Ensures display/etc. is ready before UI starts.

---

10. Bootloader CPU Frequency Initialization

- Set **big CPU** and **memory interconnect frequency** as high as safe.
- Prevents slow decompression and slow init before `cpufreq` driver loads.

⏱️ Can save **hundreds of ms**, especially on modular GKI setups.

---

11. Avoid Firmware Loading in First Stage Init

- Firmware should not be loaded by modules during early boot.
- Move to second stage if possible.

🧠 Summary Diagram (optional on request)

If you'd like a visual diagram or checklist of these boot-time optimizations, let me know — I can create that as well.

Would you like a sample `BoardConfig.mk` with all these optimizations included?

71. ↪
Probing
explain in simple

Sure! Let's break down **"probing"** and **"asynchronous probing"** in **simple terms**:

🔧 What is "probing" in Linux/Android kernel?

When the system **loads a driver module**, it **checks if any hardware** matches the driver. This check is called **"probing"**.

Think of it like this:

> 🔍 The kernel asks: "Is this driver useful for any device I have?"

If yes — it connects the driver to the device.
If no — the driver just sits there doing nothing.

🚗 Real-life example:

Imagine plugging a USB mouse into your computer.

- The kernel loads a "mouse driver" (a `.ko` module).
- The kernel **probes**: "Is this driver for this USB mouse?"
- If yes, it connects them and the mouse starts working.

⏱️ Why does probing affect boot time?

Because:

- Probing can be **slow** for some devices (e.g. I2C sensors, touchscreens, USB).
- When done during boot, **boot waits** for the driver to finish probing.

That slows down everything else.

🧵 What is asynchronous probing?

"Async probing" = do probing in the **background**, don't make the boot process wait.

Think of it like:

> 📦 "I'll start checking this driver, but you go ahead with the rest of the boot."

That way, **boot continues faster**, and the device will finish initializing when it's ready.

✅ When to use async probing?

Use it for:

- Devices on **slow buses** (like I2C, SPI)
- Drivers that load **firmware**
- Drivers that do **heavy setup**

You should NOT use async probing for:

- Devices **needed early in boot** (e.g. storage, CPUfreq, display in recovery)

🔧 How to enable async probing?

In kernel boot command line (in **`cmdline`**):

```txt
mydriver.async_probe=1
```

Or in code:

```c
driver.probe_type = PROBE_PREFER_ASYNCHRONOUS;
```

💡 Summary

Term	Simple Meaning
Probing	Driver checking if it's for a specific device
Synchronous probing	Happens during boot and blocks next steps
Asynchronous probing	Happens in the background — doesn't block boot

---

Let me know if you want a simple visual or flowchart to go with this.

72. Debug features

bookmark_border
The Linux kernel provides many mechanisms to diagnose issues including tracing, BPF, and profiling. Most of these techniques are available in the Generic Kernel Image (GKI) binary kernel. When developing products, there are many more tools that are integrated into the development or integration processes, some of which require kernel features that aren't part of GKI. As with most GKI features, we encourage you to work with upstream maintainers to add these features to Linux.

Note: Kernel debugging tools and techniques are beyond the scope of this document.
Downstream debug features in vendor modules
Debugging features are similar in scope to other kinds of features and must follow the same processes as described in Android-specific patches.

Intrusive downstream debug features
Some useful debug features require special configuration options that are disabled in GKI and therefore require a special build of the kernel. These features might be disabled due to performance penalties or unsuitability for production use, but are indispensable for debugging certain types of issues. Examples of such tools include sanitizers (KASAN, UBSAN), debugfs, and other partner-specific features and tools.

It's up to you to determine how you manage special debug-enabled kernels that impact the Kernel Module Interface (KMI). A common practice is to have an internal debug build and deliver the resulting artifacts to internal customers or downstream development partners. We recommend continuing to isolate such features into modules as partner-specific debug features can't be accepted into the GKI kernel beyond the guidelines described in Android-specific patches.

Here's a **simple explanation** of the Android kernel **debug features** section:

🔧 What are debug features in the kernel?

The Linux/Android kernel has tools built in to help developers:

- **Trace bugs**
- **Monitor performance**
- **Find memory issues**
- **Inspect kernel state (like variables, processes)**

Some of these tools are:

- **ftrace** (function tracing)
- **BPF** (advanced tracing/profiling)
- **perf** (performance analysis)
- **debugfs** (debug info via virtual filesystem)
- **KASAN/UBSAN** (memory and undefined behavior detectors)

🧠 What is GKI?

GKI (Generic Kernel Image) is a standard Android kernel built by Google that all devices must use. It includes **only stable, performance-safe features**.

🛠️ Can all debug features be used in GKI?

No. Many powerful debug tools are **not enabled** in GKI because:

- They **slow down performance**
- They **expose sensitive info**
- They are **too risky for production**

So, they are left **disabled by default**.

🧑‍🔧 What do vendors do then?

Vendors (e.g. Qualcomm, MediaTek, Samsung) often:

- **Enable these debug features** in their **own private kernel builds**
- Share those special debug builds only with:
  - Internal teams
  - Development partners
- Use them to **find and fix bugs**
- But never ship them on actual consumer devices

This way, you can:

- Keep production safe and fast ✅
- Still debug hard problems internally 🛠️

📦 Example debug-only features

Feature	Purpose
`KASAN`	Detect invalid memory access
`UBSAN`	Catch undefined behavior
`debugfs`	Expose kernel debug info
Custom logs	Vendor-specific diagnostics

✅ Summary

Concept	Explanation
Debug features	Tools to diagnose and fix kernel problems
GKI limitations	Debugging is limited in GKI due to performance and security concerns
Vendor debug builds	Vendors use separate internal builds with full debug tools
Best practice	Isolate debug tools in vendor modules, not in GKI

Let me know if you want a visual diagram to show how debug builds differ from production builds!

73. Develop kernel code for GKI

bookmark_border
The Generic Kernel Image (GKI) reduces kernel fragmentation by aligning closely with the upstream Linux kernel. However, there are valid reasons why some patches can't be accepted upstream, and there are product schedules that must be met, so some patches are maintained in the Android Common Kernel (ACK) sources from which the GKI is built.

Developers must submit code changes upstream using the Linux Kernel Mailing List (LKML) as the first choice, and submit code changes to the ACK android-mainline branch only when there's a strong reason why upstream isn't viable. Examples of valid reasons and how to handle them are listed as follows.

The patch was submitted to LKML, but wasn't accepted in time for a product release. To handle this patch:

Provide evidence that the patch was submitted to LKML and comments received for the patch, or an estimated time by which the patch is submitted upstream.
Decide on a course of action to land the patch in ACK, get it approved upstream, and then take it out of ACK when the final upstream version is merged into ACK.
The patch defines EXPORT_SYMBOLS_GPL() for a vendor module, but couldn't be submitted upstream because there are no in-tree modules that consume that symbol. To handle this patch, provide details on why your module can't be submitted upstream and the alternatives you considered before making this request.

The patch isn't generic enough for upstream and there isn't time to refactor it prior to a product release. To handle this patch, provide an estimated time by which a refactored patch is submitted upstream (the patch won't be accepted in ACK without a plan to submit a refactored patch upstream for review).

The patch can't be accepted by upstream because... <insert reason here>. To handle this patch, reach out to the Android kernel team and work with us on options to refactor the patch so that it can be submitted for review and accepted upstream.

There are plenty more potential justifications. When you submit your bug or your patch, include a valid justification and expect some iteration and discussion. We recognize that the ACK carries some patches, especially in the early phases of GKI while everyone is learning how to work upstream but can't relax product schedules to do so. Expect the upstreaming requirements to become more stringent over time.

Patch requirements
Patches must conform to the Linux kernel coding standards described in the Linux source tree, whether they're submitted upstream or to ACK. The scripts/checkpatch.pl script is run as part of Gerrit presubmit testing, so run it in advance to make sure it passes. To run the checkpatch script with the same configuration as the presubmit testing, use //build/kernel/static_analysis:checkpatch_presubmit. For details, see build/kernel/kleaf/docs/checkpatch.md.

ACK patches
Patches submitted to ACK must conform to the Linux kernel coding standards and the contribution guidelines. You must include a Change-Id tag in the commit message; if you submit the patch to multiple branches (for example, android-mainline and android12-5.4), you must use the same Change-Id for all instances of the patch.

Submit patches to LKML first for an upstream review. If the patch is:

Accepted upstream, it's merged automatically into android-mainline.
Not accepted upstream, submit it to android-mainline with a reference to the upstream submission or an explanation for why it wasn't submitted to LKML.
After a patch is accepted either upstream or in android-mainline, it can be backported to the appropriate LTS-based ACK (such as android12-5.4 and android11-5.4 for patches that fix Android-specific code). Submitting to android-mainline enables testing with new upstream release candidates and guarantees that the patch is in the next LTS-based ACK. Exceptions include cases where an upstream patch is backported to android12-5.4 (because the patch is likely to already be in android-mainline).

Upstream patches
As specified in the contribution guidelines, upstream patches destined for ACK kernels fall into the following groups (listed in order of likelihood of being accepted).

UPSTREAM: - Patches cherrypicked from 'android-mainline `are likely to be accepted into ACK if there's a reasonable use case.`
BACKPORT: - Patches from upstream that don't cherrypick cleanly and need modification are also likely to be accepted if there's a reasonable use case.
FROMGIT: - Patches cherrypicked from a maintainer branch in preparation for submitting to Linux mainline might be accepted if there's an upcoming deadline. These must be justified both for content and schedule.
FROMLIST: - Patches that have been submitted to LKML but haven't been accepted into a maintainer branch yet are unlikely to be accepted,

unless the justification is compelling enough that the patch would be accepted whether or not it lands in upstream Linux (we assume that it won't). There must be an issue associated with FROMLIST patches to facilitate discussion with the Android kernel team.
Android-specific patches
If you can't land required changes upstream, you can attempt to submit out-of-tree patches to ACK directly. Submitting out-of-tree patches requires that you create an issue in the IT that cites the patch and rationale for why the patch can't be submitted upstream (see the previous list for examples). However, there are a few cases where the code can't be submitted upstream. These cases are covered as follows and must follow the contribution guidelines for Android-specific patches and be tagged with ANDROID: prefix in the subject.

Changes to gki_defconfig
All CONFIG changes to gki_defconfig must be applied to both the arm64 and x86 versions unless the CONFIG is architecture-specific. To request a change to a CONFIG setting, create an issue in the IT to discuss the change. Any CONFIG change that affects the Kernel Module Interface (KMI) after it's frozen is rejected. In cases where partners request conflicting settings for a single config, we resolve conflicts through discussion on the related bugs.

Code that doesn't exist upstream
Modifications to code that's already Android-specific can't be sent upstream. For example, even though the binder driver is maintained upstream, modifications to priority inheritance features of the binder driver can't be sent upstream because they're Android-specific. Be explicit in your bug and patch why the code can't be sent upstream. If possible, split the patches into pieces that can be submitted upstream and Android-specific pieces that can't be submitted upstream to minimize the amount of out-of-tree code maintained in ACK.

Other changes in this category are updates to KMI representation files, KMI symbol lists, gki_defconfig, build scripts or configuration, or other scripts that don't exist upstream.

Out-of-tree modules
Upstream Linux actively discourages support for building out-of-tree modules. This is a reasonable position given that Linux maintainers don't make guarantees about in-kernel source or binary compatibility and don't want to support code that isn't in the tree. However, the GKI does make ABI guarantees for vendor modules, ensuring that KMI interfaces are stable for the supported lifetime of a kernel. Therefore, there's a class of changes to support vendor modules that are acceptable for ACK but aren't acceptable for upstream.

For example, consider a patch that adds EXPORT_SYMBOL_GPL() macros where the modules that use the export aren't in the source tree. While you must attempt to request EXPORT_SYMBOL_GPL() upstream and supply a module that uses the newly exported symbol, if there's a valid justification for why the module isn't being submitted upstream, you can submit the patch to ACK instead. You need to include the justification for why the module can't be upstreamed in the issue. (Don't request the non-GPL variant, EXPORT_SYMBOL().)

Hidden configs
Some in-tree modules automatically select hidden configs that can't be specified in gki_defconfig. For example, CONFIG_SND_SOC_TOPOLOGY is selected automatically when CONFIG_SND_SOC_SOF=y is configured. To accommodate out-of-tree module building, GKI includes a mechanism to enable hidden configs.

To enable a hidden config, add a select statement in init/Kconfig.gki so it is automatically selected based on the CONFIG_GKI_HACKS_TO_FIX kernel config, which is enabled in gki_defconfig. Use this mechanism only for hidden configs; if the config isn't hidden, it must be specified in gki_defconfig either explicitly or as a dependency.

Loadable governors
For kernel frameworks (such as cpufreq) that support loadable governors, you can override the default governor (such as cpufreq's schedutil governor. For frameworks (such as the thermal framework) that don't support loadable governors or drivers but still require a vendor-specific implementation, create an issue in the IT and consult with the Android kernel team.

We'll work with you and upstream maintainers to add the necessary support.

Vendor hooks
In past releases, you could add vendor-specific modifications directly into the core kernel. This isn't possible with GKI 2.0 because product-specific code must be implemented in modules and won't be accepted in the core kernels upstream or in ACK. To enable value-added features that partners rely on with minimal impact on core kernel code, GKI accepts vendor hooks that allow modules to be invoked from core kernel code. Additionally, key data structures can be padded with vendor data fields that are available to store vendor-specific data to implement these features.

Vendor hooks come in two variants (normal and restricted) that are based on tracepoints (not trace events) that vendor modules can attach to. For example, instead of adding a new sched_exit() function to do an accounting at task exit, vendors can add a hook in do_exit() that a vendor module can attach to for processing. An example implementation includes the following vendor hooks.

Normal vendor hooks use DECLARE_HOOK() to create a tracepoint function with the name trace_name where name is the unique identifier for the trace. By convention, normal vendor hook names begin with android_vh, so the name for sched_exit() hook would be android_vh_sched_exit.
Restricted vendor hooks are needed for cases such as scheduler hooks where the attached function must be called even if the CPU is offline or requires a nonatomic context. Restricted vendor hooks can't be detached, so modules that attach to a restricted hook can never unload.
Restricted vendor hook names begin with android_rvh.

To add a vendor hook, file an issue in IT and submit patches (as with all Android-specific patches, an issue must exist and you must provide justification). Support for vendor hooks is only in ACK, so don't send these patches to upstream Linux.

Add vendor fields to structures
You can associate vendor data with key data structures by adding android_vendor_data fields using the ANDROID_VENDOR_DATA() macros. For example, to support value-added features, append fields to structures as shown in the following code sample.

To avoid potential conflicts between fields needed by vendors and fields needed by OEMs, OEMs must never use fields declared using ANDROID_VENDOR_DATA() macros. Instead, OEMs must use ANDROID_OEM_DATA() to declare android_oem_data fields.

```
#include <linux/android_vendor.h>
...
struct important_kernel_data {
  [all the standard fields];
  /* Create vendor data for use by hook implementations. The
   * size of vendor data is based on vendor input. Vendor data
   * can be defined as single u64 fields like the following that
   * declares a single u64 field named "android_vendor_data1" :
   */
  ANDROID_VENDOR_DATA(1);

  /*
   * ...or an array can be declared. The following is equivalent to
   * u64 android_vendor_data2[20]:
   */
  ANDROID_VENDOR_DATA_ARRAY(2, 20);

  /*
   * SoC vendors must not use fields declared for OEMs and
   * OEMs must not use fields declared for SoC vendors.
   */
  ANDROID_OEM_DATA(1);

  /* no further fields */
}
```

Define vendor hooks
Add vendor hooks to kernel code as tracepoints by declaring them using DECLARE_HOOK() or DECLARE_RESTRICTED_HOOK() and then adding them to code as a tracepoint. For example, to add trace_android_vh_sched_exit() to the existing do_exit() kernel function:

```
#include <trace/hooks/exit.h>
void do_exit(long code)
{
    struct task_struct *tsk = current;
    ...
    trace_android_vh_sched_exit(tsk);
    ...
}
```

The trace_android_vh_sched_exit() function initially checks only if something is attached. However, if a vendor module registers a handler using register_trace_android_vh_sched_exit(), the registered function is called. The handler must be aware of the context with regard to held locks, RCS state, and other factors. The hook must be defined in a header file in the include/trace/hooks directory.

For example, the following code gives a possible declaration for trace_android_vh_sched_exit() in the file include/trace/hooks/exit.h.

```
/* SPDX-License-Identifier: GPL-2.0 */
#undef TRACE_SYSTEM
#define TRACE_SYSTEM sched
#define TRACE_INCLUDE_PATH trace/hooks

#if !defined(_TRACE_HOOK_SCHED_H) || defined(TRACE_HEADER_MULTI_READ)
#define _TRACE_HOOK_SCHED_H
#include <trace/hooks/vendor_hooks.h>
/*
 * Following tracepoints are not exported in tracefs and provide a
```

```
 * mechanism for vendor modules to hook and extend functionality
 */

struct task_struct;

DECLARE_HOOK(android_vh_sched_exit,
        TP_PROTO(struct task_struct *p),
        TP_ARGS(p));

#endif /* _TRACE_HOOK_SCHED_H */

/* This part must be outside protection */
#include <trace/define_trace.h>
```

To instantiate the interfaces required for the vendor hook, add the header file with the hook declaration to drivers/android/vendor_hooks.c and export the symbols. For example, the following code completes the declaration of the android_vh_sched_exit() hook.

```
#ifndef __GENKSYMS__
/* struct task_struct */
#include <linux/sched.h>
#endif

#define CREATE_TRACE_POINTS
#include <trace/hooks/vendor_hooks.h>
#include <trace/hooks/exit.h>
/*
 * Export tracepoints that act as a bare tracehook (i.e. have no trace
 * event associated with them) to allow external modules to probe
 * them.
 */
EXPORT_TRACEPOINT_SYMBOL_GPL(android_vh_sched_exit);
```

NOTE: Data structures that are used within the hook declaration need to be fully defined in order to guarantee ABI stability. Otherwise it is unsafe to dereference the opaque pointers or use the struct in sized contexts. The include which provides the full definition of such data structures should go inside the #ifndef __GENKSYMS__ section of drivers/android/vendor_hooks.c. The header files in include/trace/hooks should not include the kernel header file with the type definitions to avoid CRC changes which break the KMI. Instead forward declare the types.

Attach to vendor hooks

To use vendor hooks, the vendor module needs to register a handler for the hook (typically done during module initialization). For example, the following code shows the module foo.ko handler for trace_android_vh_sched_exit().

```
#include <trace/hooks/sched.h>
...
static void foo_sched_exit_handler(void *data, struct task_struct *p)
{
    foo_do_exit_accounting(p);
}
...
static int foo_probe(..)
{
    ...
    rc = register_trace_android_vh_sched_exit(foo_sched_exit_handler, NULL);
    ...
}
```

Use vendor hooks from header files

To use vendor hooks from header files, you may need to update the vendor hook header file to undefine TRACE_INCLUDE_PATH to avoid build errors that indicate a trace point header file couldn't be found. For example,

```
In file included from .../common/init/main.c:111:
In file included from .../common/include/trace/events/initcall.h:74:
.../common/include/trace/define_trace.h:95:10: fatal error: 'trace/hooks/initcall.h' file not found
   95 | #include TRACE_INCLUDE(TRACE_INCLUDE_FILE)
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.../common/include/trace/define_trace.h:90:32: note: expanded from macro 'TRACE_INCLUDE'
   90 | # define TRACE_INCLUDE(system) __TRACE_INCLUDE(system)
```

```
     |                          ^~~~~~~~~~~~~~~~~~~~~~~
.../common/include/trace/define_trace.h:87:34: note: expanded from macro '__TRACE_INCLUDE'
   87 | # define __TRACE_INCLUDE(system) __stringify(TRACE_INCLUDE_PATH/system.h)
      |                                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.../common/include/linux/stringify.h:10:27: note: expanded from macro '__stringify'
   10 | #define __stringify(x...)       __stringify_1(x)
      |                                 ^~~~~~~~~~~~~~~~
.../common/include/linux/stringify.h:9:29: note: expanded from macro '__stringify_1'
    9 | #define __stringify_1(x...)     #x
      |                                 ^~
<scratch space>:14:1: note: expanded from here
   14 | "trace/hooks/initcall.h"
      | ^~~~~~~~~~~~~~~~~~~~~~~~
1 error generated.
```

To fix this type of build error, apply the equivalent fix to the vendor hook header file you're including. For additional information, refer to https://r.android.com/3066703.

```
diff --git a/include/trace/hooks/mm.h b/include/trace/hooks/mm.h
index bc6de7e53d66..039926f7701d 100644
--- a/include/trace/hooks/mm.h
+++ b/include/trace/hooks/mm.h
@@ -2,7 +2,10 @@
 #undef TRACE_SYSTEM
 #define TRACE_SYSTEM mm

+#ifdef CREATE_TRACE_POINTS
 #define TRACE_INCLUDE_PATH trace/hooks
+#define UNDEF_TRACE_INCLUDE_PATH
+#endif
```

Defining UNDEF_TRACE_INCLUDE_PATH tells include/trace/define_trace.h to undefine TRACE_INCLUDE_PATH after creating the trace points.

Core kernel features
If none of the previous techniques enable you to implement a feature from a module, then you must add the feature as Android-specific modification to the core kernel. Create an issue in the issue tracker (IT) to start the conversation.

User application programming interface (UAPI)
UAPI header files. Changes to UAPI header files must occur upstream unless the changes are to Android-specific interfaces. Use vendor-specific header files to define interfaces between vendor modules and vendor userspace code.
sysfs nodes. Don't add new sysfs nodes to the GKI kernel (such additions are valid only in vendor modules). sysfs nodes used by the SoC- and device-agnostic libraries and Java code that comprises the Android framework can be changed only in compatible ways and must be changed upstream if they're not Android-specific sysfs nodes. You can create vendor-specific sysfs nodes to be used by vendor userspace. By default, access to sysfs nodes by userspace is denied using SELinux. It's up to the vendor to add the appropriate SELinux labels to allow access by authorized vendor software.
DebugFS nodes. Vendor modules can define nodes in debugfs for debugging only (as debugfs isn't mounted during normal operation of the device).

Here's a **simple explanation** of how to **develop kernel code for GKI (Generic Kernel Image)**:

🔧 What is GKI?

GKI is a **common Android kernel** built by Google that:

- Works on many devices
- Reduces fragmentation
- Keeps **stable ABI** for vendor modules

💡 How to develop kernel code in the GKI world?

✅ Preferred way: Go Upstream (LKML)

1. **Write your patch**

2. **Submit to the Linux Kernel Mailing List (LKML)**
3. **If accepted**, it will **automatically reach Android (ACK/GKI)**

⚠️ Can't submit upstream? Use ACK

If you can't upstream it, submit to **ACK (Android Common Kernel)** only if:

- You're waiting for upstream to accept it
- It's Android-specific and not accepted upstream
- It adds **symbols for vendor modules** (e.g., `EXPORT_SYMBOL_GPL`)
- It's needed quickly for a product deadline

➡️ **You must provide justification**, like:

- Upstream thread links
- Reason why upstream won't accept it
- Plan to upstream later

📋 Patch Types (based on origin)

Prefix	Meaning
`UPSTREAM:`	Patch already in upstream Linux
`BACKPORT:`	Upstream patch modified to work on old kernel
`FROMGIT:`	Patch from maintainer git branch (not upstream yet)
`FROMLIST:`	Patch submitted to LKML, not accepted yet
`ANDROID:`	Patch is Android-specific, not going upstream

📦 Where to apply patches?

1. **ACK (`android-mainline`, `android12-5.4`, etc.)**
2. Patches must:
   - Follow **Linux coding style**
   - Use `checkpatch.pl` tool
   - Include `Change-Id` in commit message
3. Submit to `android-mainline` **first**, then backport to older versions

📁 Special Patch Categories

🧱 Android-specific patches:

- Can't go upstream (e.g., binder enhancements)
- Must follow **ANDROID:** tag
- Must open a **bug ticket** with explanation

⚙️ Config changes:

- Touching `gki_defconfig` needs justification
- Must be applied to **both arm64 and x86**, unless arch-specific
- Can't break **KMI stability**

🔒 Hidden config flags:

Some configs are selected **automatically** and can't go into `gki_defconfig`.

Use:

```c
init/Kconfig.gki
CONFIG_GKI_HACKS_TO_FIX=y
```

🧩 Vendor Modules

GKI supports vendor modules through **stable ABI**.

🔧 If your module needs an exported symbol:

- Use `EXPORT_SYMBOL_GPL()`
- Explain why the module can't go upstream
- Request export via ACK

📍 Vendor Hooks (to extend core kernel)

You can't change core kernel functions directly.
Instead, use **vendor hooks**:

Type	Use when you need
`android_vh_`	Normal trace-based hook
`android_rvh_`	Restricted hook (atomic/non-removable)

Example:

```c
DECLARE_HOOK(android_vh_sched_exit,
             TP_PROTO(struct task_struct *p),
             TP_ARGS(p));
```

Add the hook to your kernel code:

```c
trace_android_vh_sched_exit(current);
```

Register it in your module:

```c
register_trace_android_vh_sched_exit(my_handler, NULL);
```

🧵 Add Vendor Fields to Kernel Structures

You can safely add extra fields for your module via:

```c
ANDROID_VENDOR_DATA(1); // for SoC vendors
ANDROID_OEM_DATA(1);    // for OEMs
```

These don't break ABI.

🧑‍💻 UAPI rules

Interface	Allowed in GKI?	Notes
UAPI headers	Only if Android-specific	
sysfs nodes	Only from vendor modules	
debugfs nodes	Yes (only for debugging)	

✅ Summary: Best Practices

Task	Do This
Add features	Try to **upstream first** (LKML)
Use GKI ABI	Submit to **ACK** only with reason
Need kernel data access	Use **vendor hooks** or **vendor fields**
Need debug feature	Put it in a **vendor module**, not core
Use sysfs/debugfs	Only inside **vendor module**, not GKI

Let me know if you want a checklist for submitting a patch, or a visual of how patches flow from developer → ACK → upstream!

74. ↪
Vendor Hooks
in simple

Here's a **super simple explanation of Vendor Hooks** in Android GKI kernel:

🧩 What are Vendor Hooks?

Vendor hooks let **your vendor kernel modules** add extra behavior to the **core GKI kernel without modifying the GKI itself**.

🔧 Why do we need them?

You can't change the GKI core directly. But sometimes you want to:

- Track when a task exits
- Do special scheduling
- Log extra things
- Add hardware-specific features

Vendor hooks let you **plug into key points** inside the kernel safely.

🔲 How do vendor hooks work?

1. **GKI kernel defines a "hook point"** like:

```c
DECLARE_HOOK(android_vh_sched_exit, ...);
```

2. Inside GKI kernel code:

```c
trace_android_vh_sched_exit(current_task);
```

3. In your **vendor module**, you register your own function:

```c
register_trace_android_vh_sched_exit(my_handler, NULL);
```

4. When that point in kernel is hit, **your function gets called**!

✌️ Two types of hooks:

Hook Type	When to Use
`android_vh_`	Normal vendor hook
`android_rvh_`	Special one: runs even on offline CPUs or atomic context; can't be removed (for critical cases)

📦 Example Use Case

You want to run your function every time a task (process/thread) exits.

1. GKI defines:

```c
DECLARE_HOOK(android_vh_sched_exit, ...);
```

2. GKI uses:

```c
trace_android_vh_sched_exit(current);
```

3. Your module:

```c
static void my_exit_handler(void *data, struct task_struct *p) {
    printk("Task exited: %s\n", p->comm);
}

register_trace_android_vh_sched_exit(my_exit_handler, NULL);
```

💡 Key Benefits

✅ No need to touch core kernel
✅ Keeps your module separate
✅ Still works with GKI updates
✅ ABI-safe

Would you like me to generate a simple diagram for this?

75. Android kernel file system support

bookmark_border
While there are a large number of file systems with implementations in the Linux Kernel, many haven't been approved for production use in Android, and are not supported in Android.

Android's test infrastructure, OTA mechanisms, update processes, and privacy requirements depend on specific file system support. Not all file systems are suitable for use on Android devices.

For example, Android requires support for file-based encryption through fscrypt and file-based authentication through fsverity, so file systems that don't support fscrypt or fsverity aren't suitable for production use.

Low level file system support
Beginning with Android 13, userspace only works with file systems built into the GKI. Shipping a file system that doesn't have support from Google's kernel team can leave users vulnerable to security issues and isn't recommended.

The Android kernel team continues taking fixes for all file systems by snapping to upstream Long Term Support (LTS) kernels. However, the following file systems receive more frequent patches through the Android security patch release schedule:

exfat (supported in kernel 5.10 and later)
ext4
f2fs
fuse
incfs
Vfat
EROFS
The following file system is deprecated and has limited support:

sdcardfs (only supported in kernel 4.14 and earlier)
Virtual file system support
In general, virtual file systems, including the following, are supported.

debugfs
overlayfs
procfs
sysfs
tmpfs
tracefs

Here's a **simple explanation** of **Android kernel file system support**:

✅ What file systems are supported in Android?

Android supports only a few **safe and secure file systems**. These are:

File System	Use Case
ext4	Most common for internal storage (`/system`, `/data`)
f2fs	Flash-friendly file system for `/data` partition
vfat / **exfat**	Used for SD cards or USB drives
fuse	Used for user-space file systems like emulated SD card
incfs	Incremental file system for streaming APKs
EROFS	Used for read-only partitions like `/system`
debugfs, procfs, sysfs, tracefs, tmpfs	Virtual FS for debugging, system control, or temp memory

---

❌ What's not supported?

- **sdcardfs** is deprecated (used before Android 10, replaced by FUSE).
- File systems that **don't support fscrypt (encryption)** or **fsverity (auth)** are **not allowed**.

- Any FS not built into the GKI is not allowed for **user data** in Android 13+.

💡 Why is Android picky about file systems?

Because of:

- **Security** (encryption/authentication)
- **Reliability** (OTA, update process)
- **Stability** (Android's testing infra)
- **Performance** (on flash storage)

📅 Regular Patch Support (actively maintained by Google):

- `ext4`
- `f2fs`
- `exfat`
- `fuse`
- `incfs`
- `vfat`
- `EROFS`

🔐 Important: fscrypt and fsverity

Android requires:

- **fscrypt**: encrypt files on a per-file basis (important for `/data`)
- **fsverity**: verify file integrity (used in system partitions like `/system`, `/vendor`)

If a file system **doesn't support these**, it's **not suitable** for Android production use.

Let me know if you want a table diagram or visual!

76. Extend the kernel with eBPF

bookmark_border
Extended Berkeley Packet Filter (eBPF) is an in-kernel virtual machine that runs user-supplied eBPF programs to extend kernel functionality. These programs can be hooked to probes or events in the kernel and used to collect useful kernel statistics, monitor, and debug. A program is loaded into the kernel using the bpf(2) syscall and is provided by the user as a binary blob of eBPF machine instructions. The Android build system has support for compiling C programs to eBPF using simple build file syntax described in this document.

More information about eBPF internals and architecture can be found at Brendan Gregg's eBPF page.

Android includes an eBPF loader and library that loads eBPF programs at boot time.

Android BPF loader
During Android boot, all eBPF programs located at /system/etc/bpf/ are loaded. These programs are binary objects built by the Android build system from C programs and are accompanied by Android.bp files in the Android source tree. The build system stores the generated objects at /system/etc/bpf, and those objects become part of the system image.

Format of an Android eBPF C program
An eBPF C program must have the following format:

```
#include <bpf_helpers.h>

/* Define one or more maps in the maps section, for example
 * define a map of type array int -> uint32_t, with 10 entries
 */
DEFINE_BPF_MAP(name_of_my_map, ARRAY, int, uint32_t, 10);

/* this also defines type-safe accessors:
 *   value * bpf_name_of_my_map_lookup_elem(&key);
 *   int bpf_name_of_my_map_update_elem(&key, &value, flags);
 *   int bpf_name_of_my_map_delete_elem(&key);
 * as such it is heavily suggested to use lowercase *_map names.
 * Also note that due to compiler deficiencies you cannot use a type
 * of 'struct foo' but must instead use just 'foo'.  As such structs
 * must not be defined as 'struct foo {}' and must instead be
 * 'typedef struct {} foo'.
 */

DEFINE_BPF_PROG("PROGTYPE/PROGNAME", AID_*, AID_*, PROGFUNC)(..args..) {
   <body-of-code
    ... read or write to MY_MAPNAME
    ... do other things
   >
}

LICENSE("GPL"); // or other license
```

Where:

name_of_my_map is the name of your map variable. This name informs the BPF loader of the type of map to create and with what parameters. This struct definition is provided by the included bpf_helpers.h header.
PROGTYPE/PROGNAME represents the type of the program and program name. The type of the program can be any of those listed in the following table. When a type of program isn't listed, there is no strict naming convention for the program; the name just needs to be known to the process that attaches the program.

PROGFUNC is a function that, when compiled, is placed in a section of the resulting file.

kprobe	Hooks PROGFUNC onto at a kernel instruction using the kprobe infrastructure. PROGNAME must be the name of the kernel function being kprobed. Refer to the kprobe kernel documentation for more information about kprobes.
tracepoint	Hooks PROGFUNC onto a tracepoint. PROGNAME must be of the format SUBSYSTEM/EVENT. For example, a tracepoint section for attaching functions to scheduler context switch events would be SEC("tracepoint/sched/sched_switch"), where sched is the name of the trace subsystem, and sched_switch is the name of the trace event. Check the trace events kernel documentationfor more information about tracepoints.

skfilter Program functions as a networking socket filter.
schedcls Program functions as a networking traffic classifier.
cgroupskb, cgroupsock Program runs whenever processes in a CGroup create an AF_INET or AF_INET6 socket.
Additional types can be found in the Loader source code.

For example, the following myschedtp.c program adds information about the latest task PID that has run on a particular CPU. This program achieves its goal by creating a map and defining a tp_sched_switch function which can be attached to the sched:sched_switch trace event. For more information, see Attaching programs to tracepoints.

```
#include <linux/bpf.h>
#include <stdbool.h>
#include <stdint.h>
#include <bpf_helpers.h>

DEFINE_BPF_MAP(cpu_pid_map, ARRAY, int, uint32_t, 1024);

struct switch_args {
    unsigned long long ignore;
    char prev_comm[16];
    int prev_pid;
    int prev_prio;
    long long prev_state;
    char next_comm[16];
    int next_pid;
    int next_prio;
};

DEFINE_BPF_PROG("tracepoint/sched/sched_switch", AID_ROOT, AID_SYSTEM, tp_sched_switch)
(struct switch_args *args) {
    int key;
    uint32_t val;

    key = bpf_get_smp_processor_id();
    val = args->next_pid;

    bpf_cpu_pid_map_update_elem(&key, &val, BPF_ANY);
    return 1; // return 1 to avoid blocking simpleperf from receiving events
}

LICENSE("GPL");
```

The LICENSE macro is used to verify if the program is compatible with the kernel's license when the program makes use of BPF helper functions provided by the kernel. Specify the name of your program's license in string form, such as LICENSE("GPL") or LICENSE("Apache 2.0").

Format of the Android.bp file
For the Android build system to build an eBPF .c program, you must create an entry in the Android.bp file of the project. For example, to build an eBPF C program named bpf_test.c, make the following entry in your project's Android.bp file:

```
bpf {
    name: "bpf_test.o",
    srcs: ["bpf_test.c"],
    cflags: [
        "-Wall",
        "-Werror",
    ],
}
```

This entry compiles the C program resulting in the object /system/etc/bpf/bpf_test.o. On boot, the Android system automatically loads the bpf_test.o program into the kernel.

Note: Your GPL-licensed eBPF C program should go in system/bpfprogs.
Files available in sysfs
During boot, the Android system automatically loads all the eBPF objects from /system/etc/bpf/, creates the maps that the program needs, and pins the loaded program with its maps to the BPF file system. These files can then be used for further interaction with the eBPF program or reading maps. This section describes the conventions used for naming these files and their locations in sysfs.

The following files are created and pinned:

For any programs loaded, assuming PROGNAME is the name of the program and FILENAME is the name of the eBPF C file, the Android loader creates and pins each program at /sys/fs/bpf/prog_FILENAME_PROGTYPE_PROGNAME.

For example, for the previous sched_switch tracepoint example in myschedtp.c, a program file is created and pinned to /sys/fs/bpf/prog_myschedtp_tracepoint_sched_sched_switch.

For any maps created, assuming MAPNAME is the name of the map and FILENAME is the name of the eBPF C file, the Android loader creates and pins each map to /sys/fs/bpf/map_FILENAME_MAPNAME.

For example, for the previous sched_switch tracepoint example in myschedtp.c, a map file is created and pinned to /sys/fs/bpf/map_myschedtp_cpu_pid_map.

bpf_obj_get() in the Android BPF library returns a file descriptor from the pinned /sys/fs/bpf file. This file descriptor can be used for further operations, such as reading maps or attaching a program to a tracepoint.

Android BPF library
The Android BPF library is named libbpf_android.so and is part of the system image. This library provides the user with low-level eBPF capabilities needed for creating and reading maps, creating probes, tracepoints, and perf buffers.

Attach programs to tracepoints
Tracepoint programs are loaded automatically at boot. After loading, the tracepoint program must be activated using these steps:

Call bpf_obj_get() to obtain the program fd from the pinned file's location. For more information, refer to the Files available in sysfs.
Call bpf_attach_tracepoint() in the BPF library, passing it the program fd and the tracepoint name.
The following code sample shows how to to attach the sched_switch tracepoint defined in the previous myschedtp.c source file (error checking isn't shown):

```
char *tp_prog_path = "/sys/fs/bpf/prog_myschedtp_tracepoint_sched_sched_switch";
char *tp_map_path = "/sys/fs/bpf/map_myschedtp_cpu_pid";

// Attach tracepoint and wait for 4 seconds
int mProgFd = bpf_obj_get(tp_prog_path);
int mMapFd = bpf_obj_get(tp_map_path);
int ret = bpf_attach_tracepoint(mProgFd, "sched", "sched_switch");
sleep(4);

// Read the map to find the last PID that ran on CPU 0
android::bpf::BpfMap<int, int> myMap(mMapFd);
printf("last PID running on CPU %d is %d\n", 0, myMap.readValue(0));
```

Read from the maps
BPF maps support arbitrary complex key and value structures or types. The Android BPF library includes an android::BpfMap class that makes use of C++ templates to instantiate BpfMap based on the key and value type for the map in question. The previous code sample demonstrates using a BpfMap with key and value as integers. The integers can also be arbitrary structures.

Thus the templatized BpfMap class lets you define a custom BpfMap object suitable for the particular map. The map can then be accessed using the custom-generated functions, which are type aware, resulting in cleaner code.

For more information about BpfMap, refer to the Android sources.

Debug issues
During boot time, several messages related to BPF loading are logged. If the loading process fails for any reason, a detailed log message is provided in logcat. Filtering the logcat logs by bpf prints all the messages and any detailed errors during load time, such as eBPF verifier errors.

Examples of eBPF in Android
The following programs in AOSP provide additional examples of using eBPF:

The netd eBPF C program is used by the networking daemon (netd) in Android for various purposes such as socket filtering and statistics gathering. To see how this program is used, check the eBPF traffic monitor sources.

The time_in_state eBPF C program calculates the amount of time an Android app spends at different CPU frequencies, which is used to calculate power.

In Android 12, the gpu_mem eBPF C program tracks total GPU memory usage for each process and for the entire system. This program is used for GPU memory profiling.

Here's a **simple explanation of eBPF in Android**:

🧠 What is eBPF?

- **eBPF (extended Berkeley Packet Filter)** is like a **virtual machine inside the Linux kernel**.
- It lets you **run small C-like programs inside the kernel** *without changing kernel code*.
- You can use it to:
  - Monitor performance
  - Collect debug info
  - Enforce security or network policies
  - Track system events (like CPU switches or memory usage)

📱 How Android uses eBPF

- Android supports eBPF from the **boot time**.
- eBPF programs are written in C, compiled to `.o` object files, and stored in:

```swift
/system/etc/bpf/
```

- During boot, Android **auto-loads these programs** into the kernel.

🧩 Structure of an eBPF Program

Here's what an eBPF program looks like:

```c
#include <bpf_helpers.h>

DEFINE_BPF_MAP(cpu_pid_map, ARRAY, int, uint32_t, 1024);  // Create a map

struct switch_args {
  int next_pid;  // data we want to access
};

DEFINE_BPF_PROG("tracepoint/sched/sched_switch", AID_ROOT, AID_SYSTEM, my_func)
(struct switch_args *args) {
  int cpu = bpf_get_smp_processor_id();
  uint32_t pid = args->next_pid;
  bpf_cpu_pid_map_update_elem(&cpu, &pid, BPF_ANY);  // Store the PID
  return 1;
}

LICENSE("GPL");
```

📁 Android.bp (Build File)

To compile this C file using Android's build system, add this to your **`Android.bp`**:

```bp
bpf {
  name: "myschedtp.o",
  srcs: ["myschedtp.c"],
  cflags: ["-Wall", "-Werror"],
}
```

📌 Where are eBPF programs/maps stored?

After boot:

- Programs go to:
  `/sys/fs/bpf/prog_<filename>_<progtype>_<progname>`
- Maps go to:
  `/sys/fs/bpf/map_<filename>_<mapname>`

🛰️ How to attach eBPF to a kernel event (tracepoint)

Example (in C++):

```cpp
int prog_fd = bpf_obj_get("/sys/fs/bpf/prog_myschedtp_tracepoint_sched_sched_switch");
int map_fd  = bpf_obj_get("/sys/fs/bpf/map_myschedtp_cpu_pid_map");

bpf_attach_tracepoint(prog_fd, "sched", "sched_switch");
```

You can then read the data like this:

```cpp
android::bpf::BpfMap<int, int> myMap(map_fd);
int lastPid = myMap.readValue(0);  // Read last PID that ran on CPU 0
```

🛠️ Debugging

If eBPF loading fails:

- Logs will appear in **`logcat`** (search for **`bpf`**)
- e.g. Verifier errors if program is invalid or unsafe

📌 Real eBPF Use Cases in Android

1. **netd**: socket filtering and traffic monitoring
2. **time_in_state**: measure how long a process runs at each CPU freq
3. **gpu_mem**: track GPU memory usage per process (used in Android 12+)

🧵 Summary

Feature	Purpose
eBPF program	Runs inside kernel for monitoring/tracking
Maps	Shared memory to store or exchange data
Tracepoints	Attach to system events like CPU switches
Android loader	Loads **`.o`** eBPF files from **`/system/etc/bpf`** at boot
Debug logs	Shown in **`logcat`** under tag **`bpf`**

Let me know if you want a visual diagram or end-to-end example!

77. ↪
📌 Where are eBPF programs/maps stored? -------------------------------------- After boot: * Programs go to: `/sys/fs/bpf/prog_<filename>_<progtype>_<progname>` * Maps go to: `/sys/fs/bpf/map_<filename>_<mapname>`
explain in simple

Here's a **very simple explanation**:

📌 Where does Android store eBPF files after boot?

When your Android phone boots:

1. **eBPF programs** (small pieces of kernel logic) are loaded and stored in:

```bash
/sys/fs/bpf/prog_<filename>_<type>_<function>
```

✅ Think of this as:

> "**prog_**" + name of your file + type of program + name of your function

2. **eBPF maps** (shared memory for storing data) are stored in:

```php-template
/sys/fs/bpf/map_<filename>_<mapname>
```

✅ Think of this as:

> "**map_**" + name of your file + name of your map

🔍 Example

If your C file is called **`myschedtp.c`**, and it has:

- a **tracepoint** function called **`sched_switch`**
- a **map** called **`cpu_pid_map`**

Then after boot:

- Program will be stored at:
`/sys/fs/bpf/prog_myschedtp_tracepoint_sched_sched_switch`
- Map will be stored at:
`/sys/fs/bpf/map_myschedtp_cpu_pid_map`

These files let other programs **access, read, or interact** with your eBPF logic running inside the kernel.

Let me know if you'd like a diagram for this!

78. still confusing

Let me simplify it **even more** — imagine you're adding a small **program inside the Android kernel** to watch something like CPU usage. Here's how it works, step by step:

🧠 What is eBPF?

eBPF = a way to run **tiny programs inside the Linux/Android kernel** without changing the kernel code.
These programs can:

- Watch what's happening (like CPU switches, network usage)
- Collect data
- Help debug or monitor performance

🛠️ How it works in Android

1. You write a tiny program in C

Example:

```c
DEFINE_BPF_PROG("tracepoint/sched/sched_switch", AID_ROOT, AID_SYSTEM, my_trace_func) {
   // Save which process just ran on which CPU
}
```

2. Android compiles it into a `.o` file (an object file)

This file is stored in:

```swift
/system/etc/bpf/
```

🔁 What happens at boot time?

When Android starts:

1. It finds your `.o` file in `/system/etc/bpf/`
2. It loads it into the kernel
3. It pins it (makes it always available) in special **/sys/fs/bpf/** files

---

🗂️ Where are these files placed?

- If your program is in `myschedtp.c`, and it tracks `sched_switch`:
  - The **program** file is pinned here:

    ```swift
    /sys/fs/bpf/prog_myschedtp_tracepoint_sched_sched_switch
    ```

  - The **map** (used to store data like CPU→PID) is pinned here:

    ```swift
    /sys/fs/bpf/map_myschedtp_cpu_pid_map
    ```

🧪 Example in plain words

Imagine this program tracks **which process ran last on CPU 0**.

Later, you can write another app that:

- Opens `/sys/fs/bpf/map_myschedtp_cpu_pid_map`
- Reads the value for CPU 0
- Prints: "Last process on CPU 0 was PID 1234"

📚 Summary

Step	What happens
1.	You write a small C file using eBPF
2.	It gets compiled to `.o` file
3.	Android loads it at boot
4.	Program & map are stored in `/sys/fs/bpf/`
5.	Other code can use them to monitor/debug

Let me know if you want a picture/diagram to make it even clearer!

79. ↪
📁 Where are these files placed? * If your program is in `myschedtp.c`, and it tracks `sched_switch`: * The **program** file is pinned here: swift CopyEdit `/sys/fs/bpf/prog_myschedtp_tracepoint_sched_sched_switch` * The **map** (used to store data like CPU→PID) is pinned here: swift CopyEdit `/sys/fs/bpf/map_myschedtp_cpu_pid_map`
confused here

No problem — let's clear up the confusion about:

📁 Where are these files placed?

When Android **boots**, it **loads eBPF programs and maps** from **`/system/etc/bpf/`**.
After loading, it places them in a **special virtual filesystem** at:

```swift
/sys/fs/bpf/
```

Now let's break down the naming with an **example**:

🔧 Say you have a file:

```c
// Filename: myschedtp.c

// eBPF program type: tracepoint
// Tracepoint: sched:sched_switch
// Map name: cpu_pid_map
```

Android will generate:

✅ 1. Program File:

- This is your eBPF program (the logic)
- Android **auto-names** it:

```swift
/sys/fs/bpf/prog_myschedtp_tracepoint_sched_sched_switch
```

🔍 Why this name?

- **`prog_`** → it's a program
- **`myschedtp`** → your source filename
- **`tracepoint_sched_sched_switch`** → the tracepoint it attaches to

✅ 2. Map File:

- This is your map used to store values (like CPU ID → last PID)
- Android auto-names it:

```swift
/sys/fs/bpf/map_myschedtp_cpu_pid_map
```

🔍 Why this name?

- **`map_`** → it's a map

- `myschedtp` → source filename
- `cpu_pid_map` → name of the map you defined in code

🧠 Summary

Think of `/sys/fs/bpf/` like a **storage area** for loaded eBPF programs and maps.

What it is	Example path
eBPF program	`/sys/fs/bpf/prog_myschedtp_tracepoint_sched_sched_switch`
eBPF data map	`/sys/fs/bpf/map_myschedtp_cpu_pid_map`

If you tell me your `.c` file and map names, I can help predict exactly what paths will be created. Want that?

80. Use DebugFS in Android 12

bookmark_border
Devices that launch with Android 12 using kernel versions higher than v5.4 are required to ship with the GKI kernel. So that partners can access DebugFS in userdebug builds while they develop on the GKI kernel, the kernel config CONFIG_DEBUG_FS is enabled in the GKI defconfig. Never mount DebugFS in user builds for devices launching on Android 12.

Userdebug builds have better test coverage than user builds and get heavily tested throughout the development cycle. The following plan minimizes the difference between the two build types with respect to DebugFS access, and provides these benefits:

Prevents userdebug builds from accidentally depending on DebugFS for new functionality
Ensures that any existing functionality that's broken by the lack of DebugFS is known early in the development cycle
Debugfs accesses in userdebug builds are categorized as follows:

DebugFS file initializations during device boot, such as a write access to a file in DebugFS to turn on debug data collection.
Bugreport generation: The dumpstate HAL reads DebugFS files when DumpstateBoard() is invoked by dumpstate. This information becomes part of the bug report.
Device-specific testing and validation.
The following table describes how each of these three categories is supported in Android 12. Note that the following only applies to userdebug builds since DebugFS can't be mounted in user builds.

Use case	Android 12 userdebug build
One-time DebugFS files initialization, during startup. This access happens only once during boot time. Dumpstate HAL performs this during HAL initialization. To enable the same, init mounts DebugFS in userdebug builds before the HAL initializes. Init calls umount() on DebugFS when the device has completed booting.
Bugreport generation: The dumpstate HAL reads DebugFS files, which become part of the bug report. Done by dumpstate HAL within DumpstateBoard() when invoked by dumpstate (DumpstateDevice.cpp). The dumpstate tool (part of the Android framework) ensures that DebugFS mounts during the invocation.
Device-specific testing and validation	Adb root and shell. Mount DebugFS from the adb shell with root access1.
1To mount DebugFS from adb shell with root access, use this command:

adb shell mount -t debugfs debugfs /sys/kernel/debug.

Note:
Android 12: There aren't any new APIs for accessing the dumpstate HAL.
Android 12: New clients don't have access to the dumpstate HAL.
Only the dumpstate HAL process and vendor-modprobe tool can access DebugFS for Android 12 launching devices, since the kernel modules it loads might create DebugFS files.
Required partner actions
Partners must enact the following based on these changes in Android 12 devices:

Make all boot time initializations of DebugFS nodes happen during the dumpstate HAL initialization. For an example of how to do this, see DNM: Example for boot time initialization of DebugFS files.
Don't allow DebugFS access during runtime. The following exceptions apply:
Bugreport generation (comes from the dumpstate HAL)
Testing and validation (accessible by adb root and shell - ensure that DebugFS is mounted first)
Developers can set the debug persistent property persist.dbg.keep_debugfs_mounted to keep DebugFs mounted across reboots on userdebug and eng builds.

GTS compliance tests ensure that the DebugFS filesystem isn't mounted in user builds. Sepolicy neverallow statements ensure that in devices launching on Android 12 or higher, unauthorized processes aren't provided access to DebugFs.

Was this helpful?

Here's a **simple explanation** of how **DebugFS** is used in **Android 12** with **GKI kernels (5.4+)**:

🧠 What is DebugFS?

- **`debugfs`** is a special file system used by the Linux kernel for **debugging and diagnostics**.
- It exposes kernel internal states via files (e.g., **`/sys/kernel/debug/`**).

🚫 In User builds (production):

- **Never use DebugFS**.
- It is **not mounted**.
- Enforced by **SELinux** + **GTS compliance tests**.

✅ In Userdebug builds (for development and testing):

DebugFS **is allowed**, but **tightly controlled**:

🟢 1. Startup initialization (boot time):

- Some debug features might require writing to DebugFS files during **boot**.
- Android's **`init`** system mounts **`debugfs`** **temporarily**, lets the **dumpstate HAL** do its work, then **unmounts** it.

🟢 2. Bugreport generation:

- When you run a bugreport (**`adb bugreport`**), the **`dumpstate HAL`** temporarily mounts **`debugfs`** to collect extra logs.
- Only the **`dumpstate HAL`** and **`vendor-modprobe`** tool are allowed to do this.

🟢 3. Manual testing:

- As a developer with **adb root access**, you can **manually mount** debugfs:

  ```bash
  adb root
  adb shell mount -t debugfs debugfs /sys/kernel/debug
  ```

- This is only allowed in **userdebug** or **eng builds**.

🔧 Dev Tip:

To **keep debugfs mounted** across reboots in userdebug/eng builds:

```bash
adb shell setprop persist.dbg.keep_debugfs_mounted 1
```

✅ Summary Table

Use Case	Allowed in Userdebug?	Notes
Boot-time DebugFS access	✅ Yes	Done by dumpstate HAL, auto-mounted and unmounted
Bugreport (dumpstate HAL)	✅ Yes	DebugFS auto-mounted during bugreport generation
Manual testing with adb shell	✅ Yes	Use **`adb root`** and manually mount debugfs
Runtime access for normal apps	❌ No	Forbidden, enforced by SELinux
Any access in User build	❌ No	Forbidden, enforced by GTS & SELinux

---

Let me know if you want a diagram or real code example for how the HAL handles DebugFS.

81. FIPS 140-3 certifiable GKI crypto module

bookmark_border
The GKI kernel includes a Linux kernel module called fips140.ko that complies with FIPS 140-3 requirements for cryptographic software modules. This module can be submitted for FIPS certification if the product running the GKI kernel requires it.

The following FIPS 140-3 requirements in particular must be fulfilled before the crypto routines may be used:

The module must check its own integrity before making cryptographic algorithms available.
The module must exercise and verify its approved cryptographic algorithms using known-answer self-tests before making them available.
Why a separate kernel module
FIPS 140-3 validation is based on the idea that once a software or hardware based module has been certified, it is never changed. If changed, it must be recertified. This does not readily match the software development processes in use today, and as a result of this requirement, FIPS software modules are generally designed to be as tightly focused on the cryptographic components as possible, to ensure that changes that are not related to cryptography don't require a reevaluation of the cryptography.

The GKI kernel is intended to be regularly updated during its entire supported lifespan. This makes it infeasible for the whole kernel to be within the FIPS module boundary, as such a module would need to be recertified upon every kernel update. Defining the "FIPS module" to be a subset of the kernel image would mitigate this problem but wouldn't solve it, as the binary contents of the "FIPS module" would still change much more frequently than needed.

Before kernel version 6.1, another consideration was that GKI was compiled with LTO (Link Time Optimization) enabled, since LTO was a prerequisite for Control Flow Integrity which is an important security feature.

Therefore, all code that is covered by the FIPS 140-3 requirements is packaged up into a separate kernel module fips140.ko which only relies on stable interfaces exposed by the GKI kernel source that it was built from. This means that the module can be used with different GKI releases of the same generation, and that it must be updated and resubmitted for certification only if any issues were fixed in the code that is carried by the module itself.

When to use the module
The GKI kernel itself carries code that depends on the crypto routines that are also packaged into the FIPS 140-3 kernel module. Therefore, the built-in crypto routines are not actually moved out of the GKI kernel but rather are copied into the module. When the module is loaded, the built-in crypto routines are deregistered from the Linux CryptoAPI and superseded by the ones carried by the module.

This means that the fips140.ko module is entirely optional, and it only makes sense to deploy it if FIPS 140-3 certification is a requirement. Beyond that, the module provides no additional capabilities, and loading it unnecessarily is only likely to impact boot time, without providing any benefit.

How to deploy the module
The module can be incorporated into the Android build using the following steps:

Add the module name to BOARD_VENDOR_RAMDISK_KERNEL_MODULES. This causes the module to be copied to the vendor ramdisk.
Add the module name to BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD. This causes the module name to be added to modules.load on the target. modules.load holds the list of modules that are loaded by init when the device boots.
The integrity self check
The FIPS 140-3 kernel module takes the HMAC-SHA256 digest of its own .code and .rodata sections at module load time, and compares it to the digest recorded in the module. This takes place after the Linux module loader has already made the usual modifications such as ELF relocation processing and alternatives patching for CPU errata to those sections. The following additional steps are taken to ensure that the digest can be reproduced correctly:

ELF relocations are preserved inside the module so that they can be applied in reverse to the input of the HMAC.
The module reverses any code patches that were made by the kernel for Dynamic Shadow Call Stack. Specifically, the module replaces any instructions that push or pop from the shadow call stack with the Pointer Authentication Code (PAC) instructions that were present originally.
All other code patching is disabled for the module, including static keys and therefore tracepoints as well as vendor hooks.
The known-answer self tests
Any implemented algorithms that are covered by the FIPS 140-3 requirements must perform a known-answer self test before being used. According to the FIPS 140-3 Implementation Guidance 10.3.A, a single test vector per algorithm using any of the supported key lengths is sufficient for ciphers, as long as both encryption and decryption are tested.

The Linux CryptoAPI has a notion of algorithm priorities, where several implementations (such as one using special crypto instructions, and a fallback for CPUs that don't implement those instructions) of the same algorithm may co-exist. Hence, there is a need to test all implementations of the same algorithm. This is necessary because the Linux CryptoAPI permits the priority based selection to be sidestepped, and for a lower-priority algorithm to be selected instead.

Algorithms included in the module
All algorithms that are included in the FIPS 140-3 module are listed as follows. This applies to the android12-5.10, android13-5.10, android13-5.15, android14-5.15, android14-6.1, and android15-6.6 kernel branches, though differences between kernel versions are noted where appropriate.

Note: Not all algorithms are relevant for FIPS 140-3 certification. Some algorithms are included because the source file that defines them also defines algorithms that are relevant. For maintainability reasons, the FIPS 140-3 module is built from unmodified GKI kernel sources.
Algorithm Implementations Approvable Definition
aes aes-generic, aes-arm64, aes-ce, AES library Yes Plain AES block cipher, with no mode of operation: All key sizes (128 bits, 192 bits, and 256 bits) are supported. All implementations other than the library implementation can be composed with a mode of operation through a template.
cmac(aes) cmac (template), cmac-aes-neon, cmac-aes-ce Yes AES-CMAC: All AES key sizes are supported. The cmac template can be composed with any implementation of aes using cmac(<aes-impl>). The other implementations are standalone.
ecb(aes) ecb (template), ecb-aes-neon, ecb-aes-neonbs, ecb-aes-ce Yes AES-ECB: All AES key sizes are supported. The ecb template can be composed with any implementation of aes using ecb(<aes-impl>). The other implementations are standalone.
cbc(aes) cbc (template), cbc-aes-neon, cbc-aes-neonbs, cbc-aes-ce Yes AES-CBC: All AES key sizes are supported. The cbc template can be composed with any implementation of aes using ctr(<aes-impl>). The other implementations are standalone.
cts(cbc(aes)) cts (template), cts-cbc-aes-neon, cts-cbc-aes-ce Yes AES-CBC-CTS or AES-CBC with ciphertext stealing: The convention used is CS3; the final two ciphertext blocks are swapped unconditionally. All AES key sizes are supported. The cts template can be composed with any implementation of cbc using cts(<cbc(aes)-impl>). The other implementations are standalone.
ctr(aes) ctr (template), ctr-aes-neon, ctr-aes-neonbs, ctr-aes-ce Yes AES-CTR: All AES key sizes are supported. The ctr template can be composed with any implementation of aes using ctr(<aes-impl>). The other implementations are standalone.
xts(aes) xts (template), xts-aes-neon, xts-aes-neonbs, xts-aes-ce Yes AES-XTS: In kernel version 6.1 and lower, all AES key sizes are supported; in kernel version 6.6 and higher, only AES-128 and AES-256 are supported. The xts template can be composed with any implementation of ecb(aes) using xts(<ecb(aes)-impl>). The other implementations are standalone. All implementations implement the weak key check required by FIPS; that is, XTS keys whose first and second halves are equal are rejected.
gcm(aes) gcm (template), gcm-aes-ce No1 AES-GCM: All AES key sizes are supported. Only 96-bit IVs are supported. As with all other AES modes in this module, the caller is responsible for providing the IVs. The gcm template can be composed with any implementations of ctr(aes) and ghash using gcm_base(<ctr(aes)-impl>,<ghash-impl>). The other implementations are standalone.
sha1 sha1-generic, sha1-ce Yes SHA-1 cryptographic hash function
sha224 sha224-generic, sha224-arm64, sha224-ce Yes SHA-224 cryptographic hash function: The code is shared with SHA-256.
sha256 sha256-generic, sha256-arm64, sha256-ce, SHA-256 library Yes SHA-256 cryptographic hash function: A library interface is provided to SHA-256 in addition to the standard CryptoAPI interface. This library interface uses a different implementation.
sha384 sha384-generic, sha384-arm64, sha384-ce Yes SHA-384 cryptographic hash function: The code is shared with SHA-512.
sha512 sha512-generic, sha512-arm64, sha512-ce Yes SHA-512 cryptographic hash function
sha3-224 sha3-224-generic Yes SHA3-224 cryptographic hash function. Only present in kernel version 6.6 and higher.
sha3-256 sha3-256-generic Yes Same as preceding, but with 256-bit digest length (SHA3-256). All digest lengths use the same Keccak implementation.
sha3-384 sha3-384-generic Yes Same as preceding, but with 384-bit digest length (SHA3-384). All digest lengths use the same Keccak implementation.
sha3-512 sha3-512-generic Yes Same as preceding, but with 512-bit digest length (SHA3-512). All digest lengths use the same Keccak implementation.
hmac hmac (template) Yes HMAC (Keyed-Hash Message Authentication Code): The hmac template can be composed with any SHA algorithm or implementation using hmac(<sha-alg>) or hmac(<sha-impl>).
stdrng drbg_pr_hmac_sha1, drbg_pr_hmac_sha256, drbg_pr_hmac_sha384, drbg_pr_hmac_sha512 Yes HMAC_DRBG instantiated with the named hash function and with prediction resistance enabled: Health checks are included. Users of this interface get their own DRBG instances.
stdrng drbg_nopr_hmac_sha1, drbg_nopr_hmac_sha256, drbg_nopr_hmac_sha384, drbg_nopr_hmac_sha512 Yes Same as the drbg_pr_* algorithms, but with prediction resistance disabled. The code is shared with the prediction-resistant variant. In kernel version 5.10, the highest-priority DRBG is drbg_nopr_hmac_sha256. In kernel version 5.15 and higher, it is drbg_pr_hmac_sha512.
jitterentropy_rng jitterentropy_rng No The Jitter RNG, either version 2.2.0 (kernel version 6.1 and lower) or version 3.4.0 (kernel version 6.6 and higher). Users of this interface get their own Jitter RNG instances. They don't reuse the instances the DRBGs use.
xcbc(aes) xcbc-aes-neon, xcbc-aes-ce No
xctr(aes) xctr-aes-neon, xctr-aes-ce No Only present in kernel version 5.15 and higher.
cbcmac(aes) cbcmac-aes-neon, cbcmac-aes-ce No
essiv(cbc(aes),sha256) essiv-cbc-aes-sha256-neon, essiv-cbc-aes-sha256-ce No
Build the module from source
For Android 14 and higher (including android-mainline), build the fips140.ko module from source using the following commands.

Build with Bazel:

```
tools/bazel run //common:fips140_dist
```

Build with build.sh (legacy):

```
BUILD_CONFIG=common/build.config.gki.aarch64.fips140 build/build.sh
```

These commands perform a full build including the kernel and the fips140.ko module with the HMAC-SHA256 digest contents embedded in it.

End user guidance
Crypto Officer guidance
To operate the kernel module, the operating system must be restricted to a single operator mode of operation. This is handled automatically by Android using memory management hardware in the processor.

The kernel module cannot be installed separately; it is included as part of the device firmware and loaded automatically on boot. It only operates in an approved mode of operation.

The Crypto Officer can cause the self-tests to be run at any time by restarting the device.

User guidance
The user of the kernel module are other kernel components that need to use cryptographic algorithms. The kernel module does not provide additional logic in the use of the algorithms, and does not store any parameters beyond the time needed to perform a cryptographic operation.

Use of the algorithms for the purposes of FIPS compliance is limited to approved algorithms. To satisfy the FIPS 140-3 "service indicator" requirement, the module provides a function fips140_is_approved_service that indicates whether an algorithm is approved.

Self test errors
In the event of a self test failure, the kernel module causes the kernel to panic and the device does not continue booting. If a reboot of the device does not resolve the issue, the device must boot into recovery mode to correct the problem by re-flashing the device.

It is expected that the module's AES-GCM implementations can be "algorithm approved" but not "module approved". They can be validated, but AES-GCM cannot be considered an approved algorithm from a FIPS module standpoint. This is because the FIPS module requirements for GCM are incompatible with GCM implementations that don't generate their own IVs. ↩️

Was this helpful?

Here's a **simple explanation** of the **FIPS 140-3 certifiable GKI crypto module (`fips140.ko`)** in Android:

🔐 What is FIPS 140-3?

- **FIPS 140-3** is a **US government security standard** for cryptographic software and hardware.
- It ensures that encryption is **secure and tested**.
- Required for **government**, **enterprise**, and **regulated industries**.

🧠 Why is `fips140.ko` needed in Android?

- The **GKI (Generic Kernel Image)** updates frequently.
- FIPS certification means **no changes** allowed after validation.
- So instead of certifying the whole kernel, Google makes a **separate crypto module** (`fips140.ko`) with only the crypto code.
- This module can be **certified once**, reused across GKI versions (as long as crypto code doesn't change).

✅ What does `fips140.ko` do?

1. **Integrity Check**
   - When loading, it verifies its own `.code` and `.rodata` sections using **HMAC-SHA256**.
   - If tampered, it **refuses to work**.
2. **Self-tests**
   - Runs **known-answer tests** for all crypto algorithms to make sure they are working correctly.
   - If a test fails, **the device crashes** (kernel panic) and won't boot.
3. **Overrides Kernel Crypto**
   - The kernel already has crypto functions.
   - When `fips140.ko` loads, it **disables** the default crypto and uses its **certified** ones.

📦 When to use it?

- **Only use it if FIPS 140-3 certification is needed.**

- If not required, don't use it — it adds **boot time** and **no extra features**.

🛤️ How to include `fips140.ko`?

1. Add module to your **vendor ramdisk**:

```make
BOARD_VENDOR_RAMDISK_KERNEL_MODULES += fips140.ko
BOARD_VENDOR_RAMDISK_KERNEL_MODULES_LOAD += fips140.ko
```

2. Build it using:

```bash
tools/bazel run //common:fips140_dist
```

or (legacy):

```bash
BUILD_CONFIG=common/build.config.gki.aarch64.fips140 build/build.sh
```

🔐 Algorithms Included (Examples):

Algorithm	Description	FIPS Approved?
AES (CBC, CTR)	Symmetric encryption (128/192/256-bit)	✅ Yes
SHA-1, SHA-2	Cryptographic hash functions	✅ Yes
HMAC	Message Authentication Code	✅ Yes
AES-GCM	Authenticated encryption	❌ Not approved[1]
DRBG	Deterministic random bit generator	✅ Yes

[1] AES-GCM isn't "approved" in FIPS context because it doesn't auto-generate IVs.

🧪 What happens if a test fails?

- **Self-test fails?**
  - The kernel **panics**, device **stops booting**.
  - You must re-flash via **recovery mode**.

🧍 Roles

- **Crypto Officer**: Ensures module loads and passes self-tests (done at boot).
- **User (Kernel/Drivers)**: Use the crypto routines; no direct control or configs.

💡 Summary

- **`fips140.ko`** is a **certifiable crypto module** for Android GKI kernels.
- Used only if **FIPS 140-3** certification is required.
- Ensures cryptographic functions are **self-checked**, **verified**, and **secure**.
- Loaded at boot, replaces built-in crypto, and **panics on failure**.

Let me know if you'd like:

- A diagram of how **`fips140.ko`** integrates into Android
- Real example with source tree paths
- Build + deployment steps for a device

82. confusing

No worries — let's break it down **very simply**.

🟢 What is `fips140.ko`?

It's a **kernel module** (a small piece of code) that handles **secure encryption** on Android.

🔒 Why does Android use it?

- Some phones (like government or enterprise) need **FIPS 140-3** certification.
- That means: **"Encryption must be tested, trusted, and unchanged."**
- Android's main kernel **keeps changing**, so it can't be FIPS-certified.
- So, Google puts all the secure crypto code into a **separate file**: `fips140.ko`.

🧪 What does this module do?

1. ✅ **Checks its own code** to make sure it hasn't been modified or hacked.
2. ✅ **Tests itself** to confirm all crypto functions work.
3. ✅ **Replaces Android's built-in encryption** with its **certified version**.
4. ❗ If the test fails, Android **crashes** to stop unsafe use.

🤔 When should you use it?

Only if:

- Your product needs **FIPS 140-3** certification.
- Example: Military phone, medical device, secure enterprise.

Otherwise:

- **Don't use it.** It adds time during boot and gives no benefit.

🧰 How is it used?

To use `fips140.ko`, you:

1. Add it to **boot config**.
2. Android **loads it during boot**.
3. It runs its **self-tests**.
4. If everything is OK, Android continues with **secure crypto**.

📌 Summary

What?	Meaning
`fips140.ko`	Crypto module for Android kernel
Why it exists	To pass **FIPS 140-3** certification (secure encryption rules)
What it does	Self-checks, self-tests, then enables crypto
When to use	Only if your device needs **government-grade security**
What if test fails?	Android stops booting (to avoid insecure behavior)

Would a diagram help you? I can generate a clear image showing this flow if you like.

83. ↪
Self-checks, self-tests, then enables crypto
why this

Great question — you're asking **why** the FIPS crypto module (`fips140.ko`) needs to do:

> **Self-checks, self-tests, then enables crypto**

Let me explain **why** these steps are **required**:

🔐 Why Self-Check?

The module first **verifies its own code** to make sure:

- It **hasn't been changed** or tampered with.
- No hacker or bug has silently modified the crypto logic.

🔍 It does this by computing a **hash (HMAC-SHA256)** of its own code and comparing it to a saved "original" value.

📌 **Why?**
FIPS 140-3 **requires this** so the module is **exactly what was certified** — no change, no tricks.

🧪 Why Self-Test?

Next, the module runs **known-answer tests** — for example:

- "If I encrypt this input with AES-128, do I get this expected result?"

It tests **all approved encryption algorithms** — like AES, SHA, HMAC — using **predefined inputs and expected outputs**.

📌 **Why?**
So we're **100% sure** the crypto functions are:

- Working correctly
- Not broken or misconfigured

If anything fails → it's **unsafe to continue** → Android **panics** (crashes to protect you).

🔓 Why Enable Crypto *After* These?

Only **if both checks pass**:

1. ✅ Code is clean
2. ✅ All crypto works correctly

...then the module allows apps or kernel to **use the secure crypto algorithms**.

📌 **Why?**
To ensure Android is **not using broken or fake encryption**, which would **violate FIPS** and risk data leaks.

🧠 Think of it like:

Step	Real World Analogy
Self-check	Checking if your bank vault door is still sealed and not broken
Self-test	Testing if your vault's lock still works with the correct key
Enable	Now allow people to put money in or take it out

Would you like a visual of this flow? I can create a diagram to help it click.

84. EROFS

bookmark_border
EROFS is a read-only file system introduced in Linux 4.19. It supports compression and deduplication, and is optimized for read performance.

The primary difference between EROFS and other compressed file systems is that it supports in-place decompression. Compressed data is stored at the end of blocks, so that it can be uncompressed into the same page. In an EROFS image, more than 99% of blocks are able to use this scheme, thus eliminating the need to allocate extra pages during read operations.

EROFS images don't have to be compressed. When using compression, however, images are around 25% smaller on average. At the highest levels of compression, images can be up to 45% smaller.

Whether using compression or not, EROFS has been shown to outperform other file systems in both random and sequential access times.

Build changes
To enable EROFS, use the file system type erofs in BoardConfig.mk. For example:

```
BOARD_SYSTEMIMAGE_FILE_SYSTEM_TYPE := erofs
BOARD_VENDORIMAGE_FILE_SYSTEM_TYPE := erofs
BOARD_PRODUCTIMAGE_FILE_SYSTEM_TYPE := erofs
BOARD_SYSTEM_EXTIMAGE_FILE_SYSTEM_TYPE := erofs
BOARD_VENDOR_DLKMIMAGE_FILE_SYSTEM_TYPE := erofs
BOARD_SYSTEM_DLKMIMAGE_FILE_SYSTEM_TYPE := erofs
```

fstab changes
The fstab type is erofs, and the only mount option needed is ro. To keep the ability to test EXT4-based GSI images, you can use two fstab entries for /system.

For example:

```
system    /system    erofs   ro             wait,slotselect,avb=vbmeta_system,logical,first_stage_mount
system    /system    ext4    ro,barrier=1    wait,slotselect,avb=vbmeta_system,logical,first_stage_mount
```

Compression tuning
By default, EROFS compresses into fixed-size blocks. Compression efficacy can be increased significantly by enabling variable-length blocks. This can be configured by the following flag:

```
BOARD_EROFS_PCLUSTER_SIZE := 262144
```

This sets the maximum PCLUSTER, or variable length block size, to 262144 bytes. The number must be a multiple of 4096. There are diminishing returns at higher values, and higher values can decrease read performance depending on the device hardware.

Disable compression
By default, the compression scheme is lz4hc. To disable compression, use:

```
BOARD_EROFS_COMPRESSOR := none
```

This can be changed on a per-partition basis as well, for example:

```
BOARD_SYSTEMIMAGE_EROFS_COMPRESSOR := none
```

Deduplication
EROFS can share duplicate blocks with the following flag:

```
BOARD_EROFS_SHARE_DUP_BLOCKS := true
```

As of Android 13, compression must be disabled to use this flag.

Impact on OTAs
As of Android 13, EROFS is fully supported with Virtual A/B. The OTA package generator can generate deltas by intelligently decompressing the LZ4 streams within the file system. As long as both the source and target builds use the same LZ4 library, the OTA package is comparable in size to an EXT4-based OTA. Even if the src or dst builds don't use the same LZ4 library, it should only have a minor impact on OTA size.

Here's a **simple explanation** of **EROFS** (Enhanced Read-Only File System) used in Android:

✅ What is EROFS?

EROFS is a **read-only file system** designed for **fast performance** and **small storage size**. It's used for system partitions like **`/system`**, **`/vendor`**, etc., in Android.

🧠 Key Benefits of EROFS:

Feature	What It Means
🔒 **Read-only**	Safer system — can't be modified accidentally or maliciously.
🗜️ **Compression**	Shrinks image size by ~25–45%, saving storage.
⚡ **In-place decompression**	It decompresses data directly into the same memory page — **faster** and uses **less RAM**.
🧹 **Deduplication**	Reuses duplicate data blocks to save space (if compression is off).
📈 **Faster reads**	Outperforms EXT4 and others in many read-heavy workloads.

🔧 How to Enable It in Android Build:

In your **`BoardConfig.mk`**:

```make
BOARD_SYSTEMIMAGE_FILE_SYSTEM_TYPE := erofs
```

Do this for all partitions you want EROFS for (like system, vendor, product, etc.)

🏁 fstab Entry Example:

```text
system /system erofs ro wait,slotselect,avb=vbmeta_system,logical,first_stage_mount
```

You can add a fallback **`ext4`** line too (for GSI testing).

⚙️ Compression Settings:

1. **Default compressor** = **`lz4hc`** (good speed + compression)
2. To disable compression:

   ```make
   BOARD_EROFS_COMPRESSOR := none
   ```

3. To set bigger compression blocks (for better ratio):

   ```make
   BOARD_EROFS_PCLUSTER_SIZE := 262144
   ```

   (means 256 KB blocks; must be a multiple of 4096)

♻️ Deduplication:

- To save space by avoiding storing the same data multiple times:

  ```make
  BOARD_EROFS_SHARE_DUP_BLOCKS := true
  ```

- Note: This only works **if compression is disabled**.

📦 OTA Support:

- Android 13+ supports EROFS with **Virtual A/B**.
- OTA tools can handle compressed files and generate delta updates efficiently.
- OTA size is **similar to EXT4-based OTA** if the same LZ4 is used.

📌 When to Use EROFS?

Use it for partitions that:

- Don't need to be modified (read-only)
- Need fast read speed (boot/system)
- Benefit from compression (save space)

Would you like a diagram of how EROFS works (in-place decompression, vs ext4)?

85. Android kernel frequently asked questions

bookmark_border
This document contains frequently asked questions about the Android kernel, known also as the Generic Kernel Image (GKI) kernel. If you are unfamiliar with GKI and GKI kernel terminology, refer to the Kernel overview.

Can Kernel Module Interface (KMI) symbol lists be changed in a stable kernel?
Changes that don't affect the existing KMI can be added to kernels that are frozen. These changes include new exported functions and symbol list entries. For details, see Manage symbol lists.

Can I change structures used by vendor modules in a stable kernel?
Structures that are part of a KMI interface can't be changed in kernels with a frozen KMI. Prior to KMI freeze, these structures can be changed.

Can I change the kernel as long as I don't break binary or source compatibility?
Changes that don't affect the KMI can be made as described in Android-specific patches.

How does GKI handle kernel configuration?
For information on kernel configuration, refer to the Changes to gki_defconfig.

How do I handle upstream bug fixes for overwrites to existing GKI modules?
If you create a vendor module based on a driver in Android Common Kernel (ACK), it's up to you to ensure that bug fixes from upstream and ACK are propagated to your module. Generally, it's in your best interest to use the GKI kernel and GKI modules without modification so these updates are provided consistently by Google.

Are there GKI specific tests?
There are Vendor Test Suite (VTS) tests that verify that a certified GKI build is installed and also to enforce per-release GKI requirements. For example, there are Android 12 VTS tests for boot header v3 and to verify the existence of required partitions in a device running a 5.10 kernel.

How do I create the project defconfig fragment file for device builds?
Many GKI-compliant devices use a config fragment to describe the configuration options needed, in addition to gki_defconfig, to build the vendor modules. Examples that are part of the ACK codebase include Cuttlefish and DB845c. There's no way to avoid some manual examination of configuration options to optimize the fragment, but scripts/diffconfig from the kernel sources is useful for comparing the results of gki_defconfig and gki_defconfig+device.fragment.

Is there a workaround for modprobe treating - and _ as equivalent?
The modprobe man page contains this description: "modprobe intelligently adds or removes a module from the Linux kernel: note that for convenience, there is no difference between _ and - in module names (automatic underscore conversion is performed)." The GKI team adheres to the standard upstream convention, so vendor tools or module naming conventions must take this adherence into account.

How do I enable debugfs for internal use?
For details on enabling debugfs, see intrusive downstream debug features.

How can I resolve mismatched ABIs between the GKI kernel with a specific module configuration enabled?
Mismatched ABIs between the GKI kernel and module configuration represents an implicit module config dependency, where enabling a module causes a binary config to be built into the resulting kernel image. Reach out to the Android kernel team (kernel-team@android.com) to determine a path forward. After the list of symbols is determined, create a bug in the Issue Tracker and upload a change to the symbol list.

Here's a **simple explanation** of the **Android Kernel (GKI) FAQ**:

🔧 1. Can I change the KMI symbol list in a stable kernel?

- ✅ **Yes**, **only to add** new symbols or functions.
- ❌ **No removing or changing** existing symbols after KMI is frozen.

🧱 2. Can I change kernel structures used by vendor modules?

- ✅ You **can** change structures **before** KMI is frozen.
- ❌ You **can't** change them **after** freeze.

🔁 3. Can I change the kernel if I don't break binary compatibility?

- ✅ Yes, **if KMI is not affected**, you're allowed to.

⚙️ 4. How is kernel configuration handled in GKI?

- GKI uses a standard base config (`gki_defconfig`).
- Device-specific configs are added **as fragments**.

🐞 5. What if I use a GKI module but also write my own similar one?

- You must **manually keep your module updated** with any bug fixes from Google.
- Best practice: **use stock GKI modules** to get updates automatically.

✅ 6. Are there tests for GKI kernels?

- Yes. **VTS tests** ensure the GKI kernel is correct and meets Android version requirements.

📄 7. How to make a config fragment for my device?

- Use a **config fragment** (a small config file) + `gki_defconfig`.
- Use the `scripts/diffconfig` tool to compare and fine-tune.

🔠 8. Why is `_` and `-` treated the same in module names?

- This is a **Linux standard behavior** in `modprobe`.
- So `my_driver-module` and `my_driver_module` are treated the same.

🐚 9. How do I use `debugfs` for development?

- See DebugFS usage guide — allowed only in `userdebug` or `eng` builds.

🧩 10. What to do if GKI kernel and modules mismatch (ABI errors)?

- This means **your module depends on something not in GKI by default**.
- Solution:
  - Contact the kernel team.
  - File a bug + request symbol whitelist changes.

Would you like this visualized as a chart or diagram?

86. Kernel networking unit tests

bookmark_border
Since Android 5.0, proper operation of the Android networking stack on Linux kernels requires a number of commits that were upstreamed relatively recently or have not yet made it upstream. It is not easy to manually verify the required kernel functionality or track the missing commits, so the Android team is sharing the tests it uses to ensure the kernel behaves as expected.

Reasons to run the tests
These tests exist for three main reasons:

The exact version of the Linux kernel used on a device is typically device-specific, and it's difficult to know whether any kernel works properly without running the tests.
Forward-porting and back-porting the kernel patches to different kernel versions or different device trees may introduce subtle issues that can be impossible to spot without running the tests.
New networking features may require new kernel functionality or kernel bug fixes.
If the tests don't pass, the device's network stack behaves incorrectly, causing user-visible connectivity bugs (such as falling off Wi-Fi networks). The device will likely also fail Android Compatibility Test Suite (CTS) tests.

Use the tests
The tests use User-Mode Linux to boot the kernel as a process on a Linux host machine. See Establishing a Build Environment for suitable operating system versions. The unit test framework boots the kernel with an appropriate disk image and runs the tests from the host file system. The tests are written in Python and use TAP interfaces to exercise kernel behaviour and the socket API.

Compile the kernel for ARCH=um
For the tests to run, the kernel must compile for ARCH=um SUBARCH=x86_64. This is a supported architecture both upstream and in the common Android kernel trees (such as android-4.4). But sometimes device kernels do not compile in this mode because device trees contain device-specific or hardware-specific code in common files (for example sys/exit.c).

In many cases, it's sufficient to ensure that hardware-specific code is behind an #ifdef. Typically this should be an #ifdef on a configuration option that controls the specific feature relevant to the code. If there is no such configuration option, put hardware-specific code inside #ifndef CONFIG_UML blocks.

In general, fixing this should be the responsibility of the kernel tree provider (such as the chipset or SoC vendor). We're working with OEMs and vendors to ensure that current and future kernels compile for ARCH=um SUBARCH=x86_64 without requiring any changes.

Run the tests
The tests are at kernel/tests/net/test. It is recommended that the tests be run from AOSP main because they are the most up-to-date; in some cases, kernel features that are necessary for proper operation in a given Android release do not yet have full test coverage in the given release. For information on how to run the tests, see the kernel network test README file. Basically, from the top of your kernel tree, run:

ANDROID_TREE/kernel/tests/net/test/run_net_test.sh all_tests.sh
Pass the tests
The kernel network test Python source files contain comments that specify kernel commits that are known to be required to pass the tests. The tests should pass in the common kernel trees - all common kernel branches android-4.4 and higher - in the kernel/common project in AOSP. Therefore, passing the tests on a kernel is simply a matter of continuously merging from the corresponding common kernel branch.

Contributions
Report issues
Report any issues with the kernel network tests in the Android issue tracker with the Component-Networking label.

Document commits and adding tests
Report issues as described above, and if possible upload a change to fix the issue, if:

The tests do not pass on the common kernel trees
You find a necessary commit that is not mentioned in the source comments,
Getting the tests to pass on upstream kernels requires major changes
You believe that the tests are overspecified, or the test fail on future kernels
You'd like to add more tests or more coverage to existing tests.

Here's a **simple explanation** of the **Android Kernel Networking Unit Tests**:

🚦 What are these tests for?

These are **automated tests** that make sure your **kernel supports all networking features** needed by Android, especially since:

- Kernels vary by device.
- Networking patches may be missing or broken.
- Android needs **specific behaviors** from the kernel to avoid bugs like:
  - Wi-Fi dropping
  - Mobile data errors
  - Failing CTS tests

✅ Why should I run these tests?

- 🧪 To **check your kernel is compatible** with Android's networking stack.
- 🔄 To **catch issues** early after porting patches or rebasing kernels.
- 🧩 To verify **new features or fixes** required for latest Android versions.

⚙️ How are the tests run?

- They use **User-Mode Linux (UML)**: a way to run your kernel **as a regular process** on a Linux PC.
- Tests use **TAP interfaces** and **Python scripts** to:
  - Send/receive packets
  - Call socket APIs
  - Validate kernel behavior

🏗️ How to prepare the kernel

You must compile the kernel for:

```bash
ARCH=um SUBARCH=x86_64
```

> UML = User Mode Linux

If the kernel has device-specific code that breaks UML build:

- Wrap that code with:

```c
#ifndef CONFIG_UML
   // hardware-specific logic
#endif
```

▶️ How to run the tests

From your kernel source root, run:

```bash
ANDROID_TREE/kernel/tests/net/test/run_net_test.sh all_tests.sh
```

> 📂 This script:
>
> - Boots the UML kernel
> - Runs all the network tests

🧠 What if tests fail?

- Your kernel is **missing required networking patches**.
- Look at the comments in the Python test files — they mention specific required kernel commits.
- Solution: **merge missing commits** from AOSP's common kernels like `android-4.4`, `android-5.10`, etc.

🧑‍🔬 Want to improve the tests?

You can:

- Report bugs in the Android issue tracker (`Component-Networking`)
- Add new tests
- Fix test assumptions or add missing commit references

Would you like a diagram or a minimal sample output from one of these tests?

87. confusing

No problem — let me explain the **Android Kernel Networking Unit Tests** in **super simple terms**, step by step:

💡 What is it?

Android needs the Linux kernel to **handle network stuff properly** (like Wi-Fi, mobile data, DNS, etc.).

To make sure your custom kernel doesn't break networking, Google made a set of **automated tests**.

🧪 What do these tests check?

They test things like:

- Can the kernel **send/receive packets** correctly?
- Does it handle **socket APIs** the right way?
- Does the kernel support the **right features and patches** for Android's networking?

🖥️ How do I run these tests?

You run them on your PC, not on the phone.

Google uses something called **User Mode Linux (UML)** — it's like running your kernel as an app on Linux instead of on a real device.

🔧 Steps to Run the Tests

1. **Compile your kernel** for UML:

```bash
ARCH=um SUBARCH=x86_64
```

2. **Run this command** from inside your Android source tree:

```bash
kernel/tests/net/test/run_net_test.sh all_tests.sh
```

3. The script will:
    - Boot your kernel in UML mode
    - Run the tests automatically
    - Show you pass/fail results

🧯 What if tests fail?

That means your kernel is **missing required networking patches**.

✅ To fix:

- Merge patches from the official Android kernels like **`android-5.10`**, **`android-5.15`**, etc.
- The test files even have comments that tell you **which patch is missing**.

🆘 Why should I care?

If your kernel fails these tests:

- Your phone might **disconnect from Wi-Fi**

- Or **fail Android certification (CTS/VTS)**
- Or cause weird network bugs

So these tests help you catch problems **before shipping**.

Would you like a 1-line summary?

> 🧪 These tests check if your Linux kernel handles Android networking correctly using simulated testing on your PC (UML).

Let me know if you want me to show a small example.

88. ransition from ION to DMA-BUF heaps (5.4 kernel only)

bookmark_border
In Android 12, GKI 2.0 replaces the ION allocator with DMA-BUF heaps for the following reasons:

Security: Because each DMA-BUF heap is a separate character device, access to each heap can be controlled separately with sepolicy. This wasn't possible with ION because allocation from any heap only required access to the /dev/ion device.
ABI stability: Unlike ION, the DMA-BUF heaps framework's IOCTL interface is ABI stable because it's maintained in the upstream Linux kernel.
Standardization: The DMA-BUF heaps framework offers a well-defined UAPI. ION allowed custom flags and heap IDs that prevented developing a common testing framework because each device's ION implementation could behave differently.
The android12-5.10 branch of the Android Common Kernel disabled CONFIG_ION on March 1, 2021.

Background
The following is a brief comparison between ION and DMA-BUF heaps.

Similarities between the ION and DMA-BUF heaps framework
The ION and DMA-BUF heaps frameworks are both heap-based DMA-BUF exporters.
They both let each heap define its own allocator and DMA-BUF ops.
Allocation performance is similar because both schemes need a single IOCTL for allocation.
Differences between the ION and DMA-BUF heaps framework
ION heaps DMA-BUF heaps
All ION allocations are done with /dev/ion. Each DMA-BUF heap is a character device that's present at /dev/dma_heap/<heap_name>.
ION supports heap private flags. DMA-BUF heaps don't support heap private flags. Each different kind of allocation is instead done from a different heap. For example, the cached and uncached system heap variants are separate heaps located at /dev/dma_heap/system and /dev/dma_heap/system_uncached.
Heap ID/mask and flags need to be specified for allocation. The heap name is used for allocation.
The following sections list the components that deal with ION and describe how to switch them over to the DMA-BUF heaps framework.

Transition kernel drivers from ION to DMA-BUF heaps
Kernel drivers implementing ION heaps
Both ION and DMA-BUF heaps allow each heap to implement its own allocators and DMA-BUF ops. So you can switch from an ION heap implementation to a DMA-BUF heap implementation by using a different set of APIs to register the heap. This table shows the ION heap registration APIs and their equivalent DMA-BUF heap APIs.

ION heaps DMA-BUF heaps
void ion_device_add_heap(struct ion_heap *heap) struct dma_heap *dma_heap_add(const struct dma_heap_export_info *exp_info);
void ion_device_remove_heap(struct ion_heap *heap) void dma_heap_put(struct dma_heap *heap);
DMA-BUF heaps don't support heap private flags. So each variant of the heap must be registered individually using the dma_heap_add() API. To facilitate code sharing, it's recommended to register all variants of the same heap within the same driver. This dma-buf: system_heap example shows the implementation of the cached and uncached variants of the system heap.

Use this dma-buf: heaps: example template to create a DMA-BUF heap from scratch.

Kernel drivers directly allocating from ION heaps
The DMA-BUF heaps framework also offers an allocation interface for in-kernel clients. Instead of specifying the heap mask and flags to select the type of allocation, the interface offered by DMA-BUF heaps takes a heap name as input.

The following shows the in-kernel ION allocation API and its equivalent DMA-BUF heap allocation APIs. Kernel drivers can use the dma_heap_find() API to query the existence of a heap. The API returns a pointer to an instance of struct dma_heap, which can then be passed as an argument to the dma_heap_buffer_alloc() API.

ION heaps DMA-BUF heaps
struct dma_buf *ion_alloc(size_t len, unsigned int heap_id_mask, unsigned int flags)
struct dma_heap *dma_heap_find(const char *name)

struct dma_buf *struct dma_buf *dma_heap_buffer_alloc(struct dma_heap *heap, size_t len, unsigned int fd_flags, unsigned int heap_flags)

Kernel drivers that use DMA-BUFs
No changes are required for drivers that import only DMA-BUFs, because a buffer allocated from an ION heap behaves exactly the same as a buffer allocated from an equivalent DMA-BUF heap.

Transition the user-space clients of ION to DMA-BUF heaps
To make the transition easy for user-space clients of ION, an abstraction library called libdmabufheap is available. libdmabufheap supports

allocation in DMA-BUF heaps and ION heaps. It first checks if a DMA-BUF heap of the specified name exists and if not, falls back to an equivalent ION heap, if one exists.

Clients should initialize a BufferAllocator object during their initialization instead of opening /dev/ion using ion_open(). This is because file descriptors created by opening /dev/ion and /dev/dma_heap/<heap_name> are managed internally by the BufferAllocator object.

To switch from libion to libdmabufheap, modify the behavior of clients as follows:

Keep track of the heap name to use for allocation, instead of the head ID/mask and heap flag.
Replace the ion_alloc_fd() API, which takes a heap mask and flag argument, with the BufferAllocator::Alloc() API, which takes a heap name instead.
This table illustrates these changes by showing how libion and libdmabufheap do an uncached system heap allocation.

Type of allocation	libion	libdmabufheap
Cached allocation from system heap	ion_alloc_fd(ionfd, size, 0, ION_HEAP_SYSTEM, ION_FLAG_CACHED, &fd)	allocator->Alloc("system", size)
Uncached allocation from system heap	ion_alloc_fd(ionfd, size, 0, ION_HEAP_SYSTEM, 0, &fd)	allocator->Alloc("system-uncached", size)

The uncached system heap variant is awaiting approval upstream but is already part of the android12-5.10 branch.

To support upgrading devices, the MapNameToIonHeap() API allows mapping a heap name to ION heap parameters (heap name or mask and flags) to let those interfaces use name-based allocations. Here is a name-based allocation example.

The documentation for every API exposed by libdmabufheap is available. The library also exposes a header file for use by C clients.

Reference Gralloc implementation
The Hikey960 gralloc implementation uses libdmabufheap, so you can use it as a reference implementation.

Required ueventd additions
For any new device-specific DMA-BUF heaps created, add a new entry to the device's ueventd.rc file. This Setup ueventd to support DMA-BUF heaps example demonstrates how this done for the DMA-BUF system heap.

Required sepolicy additions
Add sepolicy permissions to enable a userspace client to access a new DMA-BUF heap. This add required permissions example shows the sepolicy permissions created for various clients to access the DMA-BUF system heap.

Access vendor heaps from framework code
To ensure Treble compliance, framework code can only allocate from preapproved categories of vendor heaps.

Based on feedback received from partners, Google identified two categories of vendor heaps that must be accessed from framework code:

Heaps that are based on system heap with device or SoC-specific performance optimizations.
Heaps to allocate from protected memory.
Heaps based on system heap with device or SoC-specific performance optimizations
To support this use case, the heap implementation of the default DMA-BUF heap system can be overridden.

CONFIG_DMABUF_HEAPS_SYSTEM is turned off in gki_defconfig to let it be a vendor module.
VTS compliance tests ensure that the heap exists at /dev/dma_heap/system. The tests also verify that the heap can be allocated from, and that the returned file descriptor (fd) can be memory-mapped (mmapped) from user space.
The preceding points are also true for the uncached variant of the system heap, although its existence isn't mandatory for fully IO-coherent devices.

Heaps to allocate from protected memory
Secure heap implementations must be vendor-specific since the Android Common Kernel doesn't support a generic secure heap implementation.

Register your vendor-specific implementations as /dev/dma_heap/system-secure<vendor-suffix>.
These heap implementations are optional.
If the heaps exist, VTS tests ensure that allocations can be made from them.
Framework components are provided with access to these heaps so that they can enable heaps usage through the Codec2 HAL/non-binderized, same-process HALs. However, generic Android framework features can't be dependent on them due to the variability in their implementation details. If a generic secure heap implementation gets added to the Android Common Kernel in the future, it must use a different ABI to avoid conflicts with upgrading devices.
Important: To access vendor heaps from framework code, work with your Technical Account Manager at Google.
Codec 2 allocator for DMA-BUF heaps
A codec2 allocator for the DMA-BUF heaps interface is available in AOSP.

The component store interface that allows heap parameters to be specified from the C2 HAL is available with the C2 DMA-BUF heap allocator.

Sample transition flow for an ION heap

To smooth the transition from ION to DMA-BUF heaps, libdmabufheap allows switching one heap at time. The following steps demonstrate a suggested workflow for transitioning a nonlegacy ION heap named my_heap that supports one flag, ION_FLAG_MY_FLAG.

Step1: Create equivalents of the ION heap in the DMA-BUF framework. In this example, because the ION heap my_heap supports a flag ION_FLAG_MY_FLAG, we register two DMA-BUF heaps:

my_heap behavior exactly matches the behavior of the ION heap with the flag ION_FLAG_MY_FLAG disabled.
my_heap_special behavior exactly matches the behavior of the ION heap with the flag ION_FLAG_MY_FLAG enabled.
Step 2: Create the ueventd changes for the new my_heap and my_heap_special DMA-BUF heaps. At this point, the heaps are visible as /dev/dma_heap/my_heap and /dev/dma_heap/my_heap_special, with the intended permissions.

Step 3: For clients that allocate from my_heap, modify their makefiles to link to libdmabufheap. During client initialization, instantiate a BufferAllocator object and use the MapNameToIonHeap() API to map the <ION heap name/mask, flag> combination to equivalent DMA-BUF heap names.

For example:

allocator->MapNameToIonHeap("my_heap_special" /* name of DMA-BUF heap */, "my_heap" /* name of the ION heap */, ION_FLAG_MY_FLAG /* ion flags */)

Instead of using the MapNameToIonHeap() API with the name and flag parameters, you can create the mapping from <ION heap mask, flag> to equivalent DMA-BUF heap names by setting the ION heap name parameter to empty.

Step 4: Replace ion_alloc_fd() invocations with BufferAllocator::Alloc() using the appropriate heap name.

Allocation type libion libdmabufheap
Allocation from my_heap with flag ION_FLAG_MY_FLAG unset ion_alloc_fd(ionfd, size, 0, ION_HEAP_MY_HEAP, 0, &fd) allocator->Alloc("my_heap", size)
Allocation from my_heap with flag ION_FLAG_MY_FLAG set ion_alloc_fd(ionfd, size, 0, ION_HEAP_MY_HEAP,ION_FLAG_MY_FLAG, &fd) allocator->Alloc("my_heap_special", size)
At this point, the client is functional but still allocating from the ION heap because it doesn't have the required sepolicy permissions to open the DMA-BUF heap.

Step 5: Create the sepolicy permissions required for the client to access the new DMA-BUF heaps. The client is now fully equipped to allocate from the new DMA-BUF heap.

Step 6: Verify that the allocations are happening from the new DMA-BUF heap by examining logcat.

Step 7: Disable the ION heap my_heap in the kernel. If the client code doesn't need to support upgrading devices (whose kernels might only support ION heaps), you can also remove the MapNameToIonHeap() invocations.

Let me explain the **ION to DMA-BUF Heaps transition** in **simple terms**. This is a shift in how Android manages memory buffers (used for things like camera frames, graphics, video, etc.) — especially in Android 12 with the 5.4 kernel and later.

🧠 Why move from ION to DMA-BUF Heaps?

Reason	Explanation
Security	ION used a single device (`/dev/ion`) — anyone with access could allocate from any heap. DMA-BUF heaps create **separate devices per heap**, so SELinux can control access per heap.
Stability	DMA-BUF has a **stable API (UAPI)** and is maintained upstream. ION had custom flags per vendor, leading to fragmentation.
Standardization	DMA-BUF heaps have a clear structure, making testing and maintenance easier across devices.

🔁 What's the difference?

Feature	ION	DMA-BUF Heaps
Access point	`/dev/ion` (single device)	`/dev/dma_heap/<heap_name>` (separate per heap)
Flags	Needed (like `ION_FLAG_CACHED`)	No flags — you use different heap names instead
Stability	Fragile (vendor-specific flags, behavior)	Stable, upstream Linux
Memory allocator	Same concept: allocators behind heaps	Same, but better structured

---

🧩 Key Concepts

- A **heap** is just a memory pool with its own allocation behavior.
- ION heaps were all inside **`/dev/ion`**, with flags to specify behavior.
- DMA-BUF heaps are exposed directly at paths like:
  - **`/dev/dma_heap/system`**
  - **`/dev/dma_heap/system-uncached`**

🧑‍💻 Kernel Developer Changes

1. 🔧 Implement DMA-BUF heaps

ION:

```c
ion_device_add_heap(&heap);
```

DMA-BUF:

```c
dma_heap_add(&heap_info);
```

Create **separate heaps** for each behavior instead of using flags.

---

2. 💻 In-Kernel Allocation

ION:

```c
ion_alloc(size, heap_mask, flags);
```

DMA-BUF:

```c
struct dma_heap *heap = dma_heap_find("system");
dma_heap_buffer_alloc(heap, size, fd_flags, heap_flags);
```

👨‍🔧 User-Space App Changes

Use the new **`libdmabufheap`** library instead of **`libion`**.

ION:

```cpp
ion_alloc_fd(ionfd, size, 0, ION_HEAP_SYSTEM, ION_FLAG_CACHED, &fd);
```

DMA-BUF:

```cpp
BufferAllocator allocator;
allocator.Alloc("system", size);
```

🔄 Transition Plan

1. **Create DMA-BUF heap equivalents** of ION heaps
2. **Add ueventd rules** for new heaps (so **`/dev/dma_heap/<name>`** gets correct permissions)

3. **Link app or HAL to libdmabufheap**
4. **Use `MapNameToIonHeap()`** to keep compatibility on older devices
5. **Update app/driver to use `Alloc("heap_name")`**
6. **Add SELinux rules** to allow access to new `/dev/dma_heap/...`
7. **Test via logcat**
8. **Remove old ION heap code** once stable

💼 Example

Allocation	libion	libdmabufheap
cached allocation	`ION_FLAG_CACHED`	`"system"`
uncached allocation	no flag	`"system-uncached"`

🔒 Vendor Secure Heaps

To support secure buffers (e.g. for DRM or camera):

- Vendors can define secure heaps like `/dev/dma_heap/system-securevendor`.
- These are not standard — framework can access only if approved.
- Talk to Google TAM to expose them securely.

🔬 Testing/Tools

- **VTS tests** ensure `/dev/dma_heap/system` exists and works
- Use `logcat` to verify which heap is being used

Would you like:

- A visual diagram of this transition?
- A sample code snippet for a HAL or app?

89. too much confusing

No problem — let's break it down **very simply**.

🧠 What is happening?

Android used to use **ION** to allocate memory for things like graphics and camera. Now, it's switching to a better system called **DMA-BUF Heaps**.

❓Why switch?

Problem with ION	DMA-BUF Heaps Fix
All apps used one device **`/dev/ion`** → not secure	Each heap is a separate device like **`/dev/dma_heap/system`** → secure
No standard → every vendor did things differently	Standardized, upstream Linux support
Breaking changes, not stable	ABI-stable (safe for updates)

🔧 What do you need to change?

🧱 If you're in kernel code:

- Old ION: **`ion_alloc()`**
- New DMA-BUF: **`dma_heap_buffer_alloc()`** using heap name

Example:

```c
struct dma_heap *heap = dma_heap_find("system");
struct dma_buf *buf = dma_heap_buffer_alloc(heap, size, fd_flags, heap_flags);
```

📱 If you're in user space (app or HAL):

Instead of this:

```cpp
ion_alloc_fd(..., ION_HEAP_SYSTEM, ION_FLAG_CACHED, &fd);
```

You do this:

```cpp
BufferAllocator allocator;
allocator.Alloc("system", size); // or "system-uncached"
```

✅ What are the benefits?

- Security: access controlled per heap
- Stability: safer for updates
- Performance: same or better

🧾 Summary of Steps

1. Replace ION code with DMA-BUF code using heap names.
2. Update your device's permissions (ueventd, SELinux).
3. Link to new library **`libdmabufheap`**.

4. Test it.
5. Remove old ION parts.

💡 Example

Old ION Code	New DMA-BUF Code
`ion_alloc_fd(..., ION_HEAP_SYSTEM)`	`allocator.Alloc("system", size)`
`ION_FLAG_CACHED`	`"system"` heap
`ION_FLAG_UNCACHED`	`"system-uncached"` heap

Do you want:

- A **picture** of this flow?
- A **code snippet** for kernel or app?

Let me know!

90. how dma-buf heaps works

Great question! Let me explain how **DMA-BUF Heaps** work in **simple, step-by-step terms**, including:

🧱 What is DMA-BUF?

- **DMA-BUF** = a Linux framework for **sharing memory buffers** between kernel subsystems (like GPU, camera, video) and user space.
- Example: camera writes to a buffer → GPU uses it for rendering → display shows it — *without copying*.

🧰 What are DMA-BUF Heaps?

- A **newer way to allocate** those DMA-BUFs.
- Introduced to replace **ION**, which had no standard and mixed everything under one device (**`/dev/ion`**).
- **Each heap = separate device** under **`/dev/dma_heap/<heap_name>`**.

⚙️ How does DMA-BUF Heaps work?

Here's how the process flows:

1. 🧱 Heaps are defined

Each **heap** is a special allocator, like:

- **`system`** — normal RAM
- **`system-uncached`** — uncached memory
- **`secure`** — special protected memory (for DRM, camera, etc.)

Each one becomes a character device:

```sh
/dev/dma_heap/system
/dev/dma_heap/system-uncached
/dev/dma_heap/system-secure
```

---

2. 🚀 User space requests a buffer

A client (like gralloc, camera HAL, media codec, etc.) wants memory.

It does:

```cpp
int fd = open("/dev/dma_heap/system", O_RDWR);
struct dma_heap_allocation_data data = { size: 4096, fd_flags: 0, heap_flags: 0 };
ioctl(fd, DMA_HEAP_IOCTL_ALLOC, &data);
```

This gives you a **DMA-BUF fd** → a handle to the memory.

---

3. 🧠 Buffer is returned

The fd returned can now be:

- **Mapped** into user-space
- **Shared** with other drivers
- **Passed** across processes (Binder, HIDL, AIDL)

You don't copy the memory — just pass around the fd.

---

4. 🔄 Drivers use it internally

In the kernel:

```c
struct dma_heap *heap = dma_heap_find("system");
struct dma_buf *buf = dma_heap_buffer_alloc(heap, size, fd_flags, heap_flags);
```

📦 What's in a DMA-BUF heap?

Each heap implements:

- **`alloc()`** function → how to get memory
- **`dma_buf_ops`** → how to map/unmap, attach/detach, sync, etc.

Example:

- **`system_heap`** uses **`alloc_pages()`**
- **`secure_heap`** might use IOMMU-protected memory

✅ Benefits over ION

Feature	ION	DMA-BUF Heap
Access control	One global **`/dev/ion`**	Each heap is a device
Standard API	Vendor-specific hacks	Clean, upstream API
ABI stability	No	Yes
UAPI maintained?	No	Yes (mainline Linux)
Security	Weak	Strong (per-heap SELinux)

📌 Summary

- DMA-BUF heaps are **per-purpose memory allocators** in Linux.
- They **replace ION** with a **clean, modular, secure**, and **stable** API.
- Buffers are **allocated via IOCTL**, **returned as fds**, and **shared across drivers**.

Would you like a **diagram** of this or a **code example** (kernel or userspace)?

91. why so many heaps and for what purpose?

Excellent question!

The reason we have **so many DMA-BUF heaps** is to **optimize memory allocation** for **different hardware use cases** on Android devices.

Let me break it down simply:

🧱 Why multiple heaps?

Because **not all memory is equal**. Different use cases need memory with **different properties**.

Each heap provides:

- Different **performance characteristics**
- Different **caching behavior**
- Different **security levels**
- Different **physical memory types**

🧠 What are the main types of heaps and why?

Here are the most common DMA-BUF heaps and what they're used for:

Heap Name	Purpose	Characteristics
`/dev/dma_heap/system`	General-purpose memory (like `malloc`)	Normal RAM, cached, user-accessible
`/dev/dma_heap/system-uncached`	Memory for DMA hardware (e.g. ISP)	Same as above but uncached
`/dev/dma_heap/carveout`	Fixed reserved region of memory	Used for early boot or legacy systems
`/dev/dma_heap/system-secure`	Protected memory for DRM/video/camera	Not accessible to user space, encrypted/protected
`/dev/dma_heap/secure-camera`	Secure camera frames (e.g. for Face Unlock)	Only trusted camera firmware can use it
`/dev/dma_heap/secure-video`	Encrypted video playback	Used by MediaCodec, Widevine DRM
`/dev/dma_heap/custom-*`	Vendor-specific optimized heaps	Tied to SoC memory layouts or IOMMUs

📱 Examples in the real world

🟢 Normal App

- Uses `system` heap
- Needs memory that is fast and cacheable

📸 Camera HAL

- Uses `system-uncached` or `secure-camera`
- Needs uncached DMA memory for ISP or secure buffer for preview

🔒 DRM video (Widevine L1)

- Uses `system-secure` or `secure-video`
- Needs protected memory so user can't dump frames

🔐 Why not just one heap?

Using a single heap for everything:

- ❌ Wastes memory (e.g. secure memory is limited and expensive)
- ❌ Fails security (e.g. no isolation between normal and secure content)

- ❌ Causes performance issues (e.g. cacheable memory used in DMA = poor throughput)

✅ Benefits of separate heaps

1. **Security:** SEPolicy can control access per heap (`/dev/dma_heap/system-secure` can't be opened by normal apps).
2. **Performance:** Avoids using cached memory for DMA (avoids CPU ↔ device coherency bugs).
3. **Hardware Compatibility:** SoC vendors can tune each heap for different hardware (e.g. GPU, ISP, codecs).
4. **Resource Partitioning:** Separate heaps help isolate memory budgets (e.g. no stealing secure RAM for general use).

🧩 Summary

We use **multiple DMA-BUF heaps** because different clients have **different memory needs**:

- Some want speed (cached memory)
- Some want DMA-safe (uncached memory)
- Some want protection (secure memory)
- Some want IOMMU-mapped buffers
- And vendors may want custom optimizations

This gives Android **flexibility**, **security**, and **performance** — all at once.

Let me know if you'd like:

- A diagram of the heap usage across Android components
- A walkthrough of heap selection in code
- Example sepolicy rules or device tree heap registration

92. the what issue ion had

Great follow-up! Let's clearly explain **why ION was replaced** by **DMA-BUF heaps** in Android — especially in **GKI (Generic Kernel Image)** starting Android 12.

🧨 Problems with ION

Issue	Why it's a problem
🔒 **Security limitations**	ION used a **single device node `/dev/ion`**, so **any app** with access could allocate from **any heap**, even secure ones. SEPolicy couldn't restrict access per heap.
💥 **No ABI stability**	ION's API wasn't stable. Vendors used **custom flags**, **custom heaps**, and the IOCTL interface would change — breaking compatibility with Android's **modular kernel model (GKI)**.
🧪 **No UAPI standard**	ION allowed custom extensions, so behavior differed from vendor to vendor. This made it **impossible to write common test frameworks** or reusable code.
🔁 **Hard to evolve**	Google couldn't upstream ION properly, and improving it meant fighting its **design limitations**. No clean separation of heaps, flags, access control, etc.
🧱 **Inflexible heap logic**	Heaps and flags were combined using bit masks, and flags were ambiguous. Couldn't cleanly define heap behavior — everything was mixed.
🔍 **Debugging and tracing pain**	Because everything went through **`/dev/ion`**, it was hard to figure out **which client allocated what**, or trace issues like memory leaks.

⚙️ Example Problem

Say a camera app uses ION to allocate secure memory:

```cpp
ion_alloc_fd(ion_fd, size, ION_HEAP_SECURE, ION_FLAG_PROTECTED, &fd);
```

- **Issue**: Any app with **`/dev/ion`** access can try the same call.
- **SEPolicy**: Can't say "allow access only to ION heap X" — no per-heap control.
- **Stability**: If vendor changes heap IDs, all clients break.
- **Security**: Hard to guarantee that **only** trusted apps touch secure heaps.

✅ DMA-BUF Heaps Fix All That

DMA-BUF Heaps Feature	Solves Which ION Problem
One device per heap: **`/dev/dma_heap/system`**, **`/dev/dma_heap/system-secure`**, etc.	🛡️ Per-heap **access control** using SEPolicy
Stable UAPI, upstreamed in Linux	✅ Long-term **ABI stability** for GKI
No custom flags or masks	🚫 No vendor-specific behavior — **consistent across devices**
Clean IOCTLs (only alloc)	🧼 Simpler and **testable** interfaces
Separate heaps = separate drivers	🔧 Vendors can optimize and track heap usage cleanly

---

🧩 Summary

🚫 ION was:

- One giant, unsafe interface
- Full of vendor-specific hacks
- Unstable and unscalable

✅ DMA-BUF Heaps is:

- Secure
- Modular
- Standardized
- GKI-compatible

Would you like a **diagram comparing ION and DMA-BUF Heaps**, or a **sample migration in code** from ION to DMA-BUF Heaps?

93. Incremental File System

bookmark_border
The Incremental File System (IncFS) kernel module introduced in Android 11 enables the Android OS to receive streamed APKs over the Android Debug Bridge (ADB).

Important: The loadable module support feature introduced in Android 11 and designated by CONFIG_INCREMENTAL_FS=m in device.mk files has been deprecated in favor of the built-in kernel configuration. That support was removed to simplify the platform code and setup. For Android 12 and higher use the built-in kernel configuration described on this page. Instructions intended for Android 11 only are labeled as such.
This self-contained kernel module creates a new virtual file system that sits on top of the existing Android file system. This complements changes in the framework and SDK to enable app and game developers to deploy large APKs through the ADB to a device running on Android 11 or higher.

The kernel change enables a new APK Signature Scheme v4 format and supports Android framework changes in the Android Package Manager, new system services, and changes to the ADB.

Implementation
To implement the IncFS, OEMs and SoC manufacturers must add a new kernel driver to their Android device builds.

For Android 11 only, if the kernel driver is built as a module it's loaded on demand. If there aren't any apps installed through an ADB incremental installation, the device doesn't load the kernel driver.

Otherwise, when it builds as a part of the kernel image, the driver is always loaded. This implementation is valid for Android 12 and higher, and can be used with Android 11. For information about upgrading the kernel driver to Android 12, see Kernel driver upgrade.

The kernel driver is part of a larger system to enable streamed APK installations. OEMs and vendors don't need to use the exact IncFS code provided in the sample implementations. However, to ensure a consistent experience across devices, you must ensure the API implementation has a file system that has file-read functionality and directory read-write functionality as defined in the Userspace interface for Incremental FS documentation.

Additionally, implementations must have mount options and special files that functionally match the IncFS sample implementation.

The following lists the necessary changes for implementation:

Set up the development machine to build the kernel.
Target the common kernel from the common-android-mainline branch.

repo init -u https://android.googlesource.com/kernel/manifest -b common-android-mainline
repo sync
Validate that the following changes that are needed for IncFS are in the branch checkout:
https://android-review.googlesource.com/c/kernel/common/+/1222869/
https://android-review.googlesource.com/c/kernel/common/+/1222870
https://android-review.googlesource.com/c/kernel/common/+/1222871
https://android-review.googlesource.com/q/%2522ANDROID:+Incremental+fs:%2522+branch:android-mainline+status:merg
Append either CONFIG_INCREMENTAL_FS=y or for Android 11 only, CONFIG_INCREMENTAL_FS=m at the bottom of the defconfig file. To see an example, click one of the links below:
Generic kernel image
Mainline module image
Build the kernel
Embed the kernel into the Android device image build.
Supported in Android 11 only: This applies only if you're using the CONFIG_INCREMENTAL_FS=m configuration. Ensure that the incrementalfs.ko module file is a part of your modules directory (usually /vendor/lib/modules/).
For your target Android device, append one of the following vendor-specific system-property lines to your device.mk file (optional in devices launched with Android 12 and higher):
When you're using CONFIG_INCREMENTAL_FS=y, append the file with one of these:

PRODUCT_PROPERTY_OVERRIDES += \
ro.incremental.enable=yes
When you're using CONFIG_INCREMENTAL_FS=m (for Android 11 only), append the file with one of these:

PRODUCT_PROPERTY_OVERRIDES += \
ro.incremental.enable=module:/vendor/lib/modules/incrementalfs.ko
See the example device.mk files for the Android emulator and Pixel 4.

For Android 11 only: If you're using CONFIG_INCREMENTAL_FS=m, add SE Linux Rules.
Create and add a vold.te file to your device /system/sepolicy/vendor folder with the following content:

vold.te
Allow it to load the incremental file system driver:

allow vold self:capability sys_module;
allow vold vendor_incremental_module:file r_file_perms;
allow vold vendor_incremental_module:system module_load;
Append the following SE Linux rules to the existing file.te file found in your /system/sepolicy/vendor folder:

file.te file - For an example see this file.te file.)
Incremental file system driver
type vendor_incremental_module, vendor_file_type, file_type;
Append the following SE Linux rules to existing file_contents file found in your /system/sepolicy/vendor folder:

file_contents file - For an example, see this file_contents file.
Incremental file system driver
/vendor/lib/modules/incrementalfs\.ko
u:object_r:vendor_incremental_module:s0
Kernel driver upgrade
Devices upgrading to Android 12 might include an older version of the IncFS driver. For those devices, AOSP recommends that you update the IncFS driver to the current version (in this case v2) for these reasons:

The version released with Android 11 is the initial implementation of IncFS, targeted only for ADB installation support.
Android 12 uses the IncFS driver for streaming installations of Play games, which requires the new features and optimizations of IncFS v2 for a better user experience.
V1 supports game streaming, but does so with performance penalties and higher battery, CPU, and RAM usage than v2.
V2 provides improved UX for streaming, with smooth progress animations, precise disk-space usage reporting, and prevention of 3rd-party apps-streaming interference.
To upgrade the IncFS driver in your kernel, apply the following patches for either kernel 4.14 or kernel 4.19:

Kernel 4.14 patch
Kernel 4.19 patch
For all other custom kernel versions please port one of the patchsets. They only affect the fs/incfs directory and apply cleanly to the existing v1 code.

If you don’t upgrade the driver, you must apply one of the fixes listed below to the v1 driver. This is required for passing the Android 12 CTS and was released in the June QPR.
Kernel 4.14 fix to v1 driver
Kernel 4.19 fix to v1 drivr
Kernel 5.4 fix to v1 driver
Continue using the IncFS driver in the same manner as for the original but now upgraded Android 11, either as a built-in part of the kernel image, or as a separate module. Don’t change the system board or system property configuration.

New devices using a GKI kernel image get the latest (v2) IncFS driver automatically, configured as a part of the kernel image. This doesn’t require additional steps.

The loadable module configuration was deprecated in Android 12, and isn’t supported for new devices. It’s only allowed for the upgrades, or for a vendor image freeze when the original kernel already had it built as a module.

Reference implementations
This implementation can be considered either as part of a kernel image, or (for Android 11 only) as a loadable module.

Loadable module (Pixel 4 device)
Add Kernel Module Prebuilts
Add and Enable Kernel Module System Property Change on Device
Update SE Linux Rules
Android Emulator (as a part of the kernel image)
Enable IncFS kernel driver
Enable Incremental on the device
Validation and testing
Validate the implementation using Feature Unit Tests, CTS and GTS.

CTS
Use CtsIncrementalInstallHostTestCases.

```
GTS
atest GtsIncrementalInstallTestCases:


/gts-tests/tests/packageinstaller/incremental/src/com/google/android/packageinstaller/incremental/gts/IncrementalInstallerTest.java
Test the IncFS
Set up a development environment.
Complete implementation tasks outlined in the implementation section.
Run the following manual tests:

mmma system/incremental_delivery/incfs/tests

atest libincfs-test

atest IncrementalServiceTest

atest PackageManagerShellCommandTest

PackageManagerShellCommandIncrementalTest
Test IncFS with Android SDK (ADB and apksigner)
Set up a development environment.
Complete implementation tasks outlined in the implementation section.
Flash the build on a target physical device or emulator.
Generate or obtain an existing APK.
Create a debug signing key.
Sign the APK with v4 signature format from thebuild-tools folder.

./apksigner sign --ks debug.keystore game.apk
Install the APK on-device from the platform-tools folder.

./adb install game.apk
```

Here's a **clear and simplified explanation** of the **Incremental File System (IncFS)** in Android:

🧩 What is Incremental File System (IncFS)?

IncFS is a **special virtual file system** added in **Android 11+** that allows apps (especially big games or APKs) to be **used before the full download is complete** — kind of like "streaming" an app while it's still being installed.

🚀 Why was IncFS introduced?

1. **Faster App Installs**
   You can launch an app or game **before** the full APK is downloaded.
2. **ADB Streaming**
   Helps developers **push big APKs over ADB** without waiting for full transfer.
3. **Game Streaming via Play Store**
   Starting in **Android 12**, Play Store can **install and run large games immediately** using IncFS.

🧠 How it works (Simplified):

Imagine installing a 1GB game. Instead of waiting for all 1GB to download:

1. 📁 **IncFS creates a virtual directory** in the kernel where APK files are stored.
2. ⬇️ **Only the parts of the file needed right now are downloaded first.**
3. 📲 The app launches using the available data.
4. 🔄 Behind the scenes, more chunks of data stream in as the game runs.

This is similar to **streaming a movie** — you don't download it all before watching.

🛠️ Key Components:

Component	Role
`incfs.ko` (Android 11)	Kernel module version of IncFS. Loaded when needed.
Built-in IncFS (Android 12+)	IncFS is built directly into the kernel image.
`/dev/incfs`	The device node used by Android to interact with IncFS.
ADB / Play Store	Tools that stream APKs via IncFS
v4 Signature	New signature scheme to verify streamed APKs

🔄 Evolution Across Android Versions:

Android Version	How IncFS is added	Notes
Android 11	As a kernel module (**`CONFIG_INCREMENTAL_FS=m`**)	Optional loadable driver
Android 12+	Built into the kernel image (**`CONFIG_INCREMENTAL_FS=y`**)	Module approach deprecated

🔧 Developer Tasks (OEMs/Silicon Vendors):

To support IncFS:

1. **Enable it in your kernel config** (`CONFIG_INCREMENTAL_FS=y`)
2. **Make sure the kernel includes correct patches** from AOSP mainline
3. **Update SEPolicy rules** to allow access if needed
4. **Add system property**:

   ```ini
   ro.incremental.enable=yes
   ```

5. **Ensure vold can mount it** (if needed for module case)

✅ Advantages

Benefit	Description
⏱️ Faster installs	Apps can launch before complete download
💾 Less storage required initially	Only needed chunks are downloaded first
🔐 Secure	Supports signature v4 to verify streamed APKs
🔌 Works with ADB & Play Store	Unified framework for dev and user installs

🧪 How to Test It

You can test IncFS support via:

- ✅ **`adb install`** large APKs with v4 signature
- ✅ CTS: **`CtsIncrementalInstallHostTestCases`**
- ✅ GTS: **`GtsIncrementalInstallTestCases`**
- ✅ Manual AOSP test:

  ```nginx
  atest libincfs-test
  atest IncrementalServiceTest
  ```

🧰 Summary

Concept	Description
What	Virtual file system for streaming APKs
Why	Fast installs, especially for games
Introduced in	Android 11 (as module), Android 12+ (built-in)
Used by	ADB, Play Store, Game installs
Replaces	Full blocking APK install flow
Related tech	APK Signature v4, SEPolicy, `incfs.ko`

Would you like a **diagram of the IncFS architecture** or a **code-level example of how it mounts and streams**?

> 94. "Summarize all the technical topics we've discussed in this conversation in a well-structured list format with main topics and their respective subtopics. Organize the topics hierarchically and logically (e.g., Android → HAL → AIDL, Kernel → ABI → Module.symvers). Make the list comprehensive, removing duplication and grouping related concepts together."

Certainly! Here's a **comprehensive, logically structured summary** of all the technical topics we discussed in this conversation, organized into main topics and their subtopics:

---

1. Android Kernel Internals

1.1 Kernel ABI & Symbol Interfaces

- Kernel Module Interface (KMI)
  - Frozen KMI & compatibility rules
  - Symbol list management (`Module.symvers`)
- ABI stability policies
- Kernel structure usage restrictions
- Kernel bug fix backporting & vendor module maintenance

1.2 File Systems

- **EROFS (Enhanced Read-Only File System)**
  - Compression (fixed and variable block sizes)
  - Deduplication
  - OTA impact and delta generation
  - Mount options in `fstab`
- **Incremental File System (IncFS)**
  - APK streaming support via ADB/Play Store
  - Kernel configuration: `CONFIG_INCREMENTAL_FS=y` or `=m`
  - IncFS v1 vs v2 (features, performance, usage)
  - SEPolicy additions, vold, and ueventd support
  - CTS/GTS test coverage

1.3 Memory Management and Allocators

- **ION vs DMA-BUF Heaps**
  - ION limitations:
    - Global access via `/dev/ion`
    - Lack of ABI stability
    - Non-standard behavior (heap IDs, flags)
  - DMA-BUF Heaps improvements:
    - One device node per heap (`/dev/dma_heap/<heap_name>`)
    - SEPolicy granularity
    - ABI-stable ioctl interface
    - Standard user/kernel allocation APIs
  - libdmabufheap abstraction:
    - `BufferAllocator::Alloc()`
    - `MapNameToIonHeap()`
    - Heap name-based access

- Secure/vendor heap access and sepolicy control

---

2. Android Boot and System Configuration

2.1 Boot-Time Optimizations

- Fastboot/bootloader phases
- Early init and `first_stage_mount`
- EROFS mounting and system partition usage
- Kernel module loading strategies
- CPU frequency initialization and governors

2.2 Vendor Partition Configuration

- BoardConfig.mk:
  - `BOARD_SYSTEMIMAGE_FILE_SYSTEM_TYPE := erofs`
  - Other partition file system overrides
- Dynamic partition support via `fstab`

---

3. Android IPC and HAL Interfaces

3.1 HAL Interface Types

- **HIDL HAL (Deprecated)**
  - XML/IDL based
  - Used via `android.hardware.*@x.y`
- **AIDL HAL**
  - Native C++ support with NDK
  - Shared memory support (FMQ, `aidl::SharedMemory`)
  - Interfaces: `ISensorHal`, `ILed`, etc.

3.2 Shared Memory Mechanisms

- **FMQ (Fast Message Queue)**
  - Optimized for sensor/data streams
  - Synchronized and Unsynchronized variants
- **Anonymous Shared Memory (ashmem)**
  - Legacy
  - Not tied to DMA
- **Gralloc / AHardwareBuffer**
  - Buffer allocation from DMA-BUF (via `GraphicBuffer`)
  - GPU-compatible for rendering/camera use cases

---

4. DMA-BUF Heaps and Graphics Pipeline

4.1 DMA-BUF Heaps Use Cases

- System / system-uncached heap
- Secure heaps (e.g., `system-secure<vendor>`)
- Vendor-optimized variants
- libdmabufheap integration for backward compatibility

4.2 Gralloc and AHardwareBuffer

- Buffer allocation flow:
  - `AHardwareBuffer` → `GraphicBuffer` → `gralloc` → `DMA-BUF`
- Benefits:
  - Efficient zero-copy GPU buffer sharing
  - Used in camera, GPU, and display frameworks

---

5. Kernel Testing Infrastructure

5.1 Networking Unit Tests

- Purpose:
  - Ensure kernel supports Android networking stack
  - Detect subtle bugs from backporting
- Mechanism:
  - User-Mode Linux (ARCH=um)
  - TAP interfaces + Python tests
- Run using: `run_net_test.sh`

5.2 CTS & VTS Tests

- CTS:
  - `CtsIncrementalInstallHostTestCases`
- GTS:
  - `GtsIncrementalInstallTestCases`
- ABI compatibility via:
  - `vts_abi_test`
  - Symbol list validation

---

6. SELinux and Permissions

6.1 SEPolicy for Kernel Modules

- ueventd.rc entries for `/dev/dma_heap/<heap>`
- SELinux contexts:
  - `vendor_incremental_module`, `vendor_file_type`, etc.
- Permissions for vold and system services

---

7. Tools and Build System Integration

7.1 Build Configuration Flags

- Kernel defconfig flags:
  - `CONFIG_ION`
  - `CONFIG_DMABUF_HEAPS`
  - `CONFIG_INCREMENTAL_FS`
- BoardConfig.mk flags for filesystems and heaps
- Makefile integration for libdmabufheap

7.2 User-space Libraries

- `libion` (legacy)
- `libdmabufheap` (current)
- `apksigner` for v4 signatures (used in IncFS)

✅ Summary

Area	Key Technologies
Kernel ABI	KMI, Symbol lists, Module.symvers
Memory	ION, DMA-BUF, FMQ, ashmem, AHardwareBuffer
File Systems	EROFS, IncFS
HAL/IPC	HIDL, AIDL, shared memory
Boot & Init	fstab, SEPolicy, module loading
Graphics	Gralloc, DMA-BUF integration
Tests	CTS, GTS, Kernel net tests
Build System	Kernel flags, BoardConfig.mk, SEPolicy rules

Let me know if you want a **diagram**, **flowchart**, or **Markdown version** of this list!