VNDK

1. Vendor Native Development Kit (VNDK) overview

bookmark_border
The Vendor Native Development Kit (VNDK) is a set of libraries used by other libraries or binaries, in the vendor or product partition, on runtime for dlopen.

Deprecation
Vendor NDK was introduced in Android 8.0 to provide APIs between the framework and vendor code. While VNDK has been successfully used for many years, it has some drawbacks:
Storage
A single VNDK APEX packages all VNDK libraries, whether they're used from the device or not.
GSI contains multiple versions of VNDK APEXes to support multiple versions of vendor images.
Updatability
It's difficult to update VNDK APEXes separately from the platform update.
Vendor images are frequently updated over the air (OTA), reducing the benefits of having VNDK packaged within the system image.
Based on these issues, we decided to deprecate VNDK starting with Android 15.
Details on VNDK deprecation
All VNDK libraries are packaged into the VNDK APEX, and installed in the system (-ext) image. With VNDK deprecation, former VNDK libraries are installed in the vendor (or product) image, same as other vendor-available libraries. These features are removed along with VNDK deprecation:
VNDK APEX for Android 15
System properties that indicate the version of the target VNDK are removed if the vendor or product partitions are built for Android 15:
ro.vndk.version
ro.product.vndk.version
VNDK optimizations won't be available as there is no VNDK:
TARGET_VNDK_USING_CORE_VARIANT for Android Go devices
use_vndk_as_stable for Vendor APEXes
Vendor snapshot, which is highly dependent on the VNDK
Exceptions from the deprecation
These features won't change with the VNDK deprecation:
VNDK APEXes with VNDK version 14 or lower, which are required to support existing vendor images.
LL-NDK isn't part of the VNDK.
Why VNDK?
AOSP allows framework-only updates in which the system partition can be upgraded to the latest framework version while vendor partition is left unchanged. Despite being built at different times, binaries in each partition must be able to work with each other.

Framework-only updates include the following challenges:

Dependency between framework modules and vendor modules. Before Android 8.0, modules in the vendor and system partition could link with each other. However, dependencies from vendor modules imposed undesired restrictions to framework modules development.
Extensions to AOSP libraries. Android requires all Android devices to pass CTS when the system partition is replaced with a standard Generic System Image (GSI). However, as vendors extend AOSP libraries to boost performance or to add extra functionalities for their HIDL implementations, flashing the system partition with a standard GSI might break a vendor's HIDL implementation. For guidelines on preventing such breakages, see VNDK extensions.
To address these challenges, Android contains several features such as VNDK (described in this section), HIDL, hwbinder, device tree overlay, and sepolicy overlay.

VNDK-specific terms
VNDK-related documents use the following terminology:
Modules refer to either shared libraries or executables. Modules make build-time dependencies.
Processes are operating system tasks spawned from executables. Processes make run-time dependencies.
Framework-qualified terms are related to the system partition:
Framework executables refer to executables in /system/bin or /system/xbin.
Framework shared libraries refer to shared libraries under /system/lib[64].
Framework modules refer to both framework shared libraries and framework executables.
Framework processes are processes spawned from framework executables, such as /system/bin/app_process.
Vendor-qualified terms are related to vendor partitions:
Vendor executables refer to executables in /vendor/bin
Vendor shared libraries refer to shared libraries under /vendor/lib[64].
Vendor modules refer to both vendor executables and vendor shared libraries.
Vendor processes are processes spawned from Vendor Executables, such as /vendor/bin/android.hardware.camera.provider@2.4-service.
Note: Generic System Image (GSI) stands for the standard Android system image that is built from corresponding branches (similar to the release branch but with some bug fixes or some generalizations) and released by Google.
VNDK concepts

In an ideal Android 8.0 and higher world, framework processes do not load vendor shared libraries, all vendor processes load only vendor shared libraries (and a portion of framework shared libraries), and communications between framework processes and vendor processes are governed by HIDL and hardware binder.

Such a world includes the possibility that stable, public APIs from framework shared libraries might not be sufficient for vendor module developers (although APIs can change between Android releases), requiring that some portion of framework shared libraries be accessible to vendor processes. In addition, as performance requirements can lead to compromises, some response-time-critical HALs must be treated differently.

The following sections detail how VNDK handles framework shared libraries for vendors and Same-Process HALs (SP-HALs).

Framework shared libraries for vendor
This section describes the criteria for classifying shared libraries that are accessible to vendor processes. There are two approaches to support vendor modules across multiple Android releases:

Stabilize the ABIs/APIs of the framework shared libraries. New framework modules and old vendor modules can use the same shared library to reduce memory footprint and storage size. A unique shared library also avoids several double-loading issues. However, the development cost to maintain stable ABIs/APIs is high and it is unrealistic to stabilize all ABIs/APIs exported by every framework shared library.
Copy old framework shared libraries. Comes with the strong restriction against side channels, defined as all mechanisms to communicate among framework modules and vendor modules, including (but not limited to) binder, socket, pipe, shared memory, shared file, and system properties. There must be no communication unless the communication protocol is frozen and stable (e.g. HIDL through hwbinder). Double-loading shared libraries might cause problems as well; for example, if an object created by the new library is passed into the functions from the old library, an error may occur as these libraries may interpret the object differently.
Different approaches are used depending on the characteristics of the shared libraries. As a result, framework shared libraries are classified into three sub-categories:

LL-NDK Libraries are Framework Shared Libraries that are known to be stable. Their developers are committed to maintain their API/ABI stabilities.
LL-NDK includes the following libraries: libEGL.so, libGLESv1_CM.so, libGLESv2.so, libGLESv3.so, libandroid_net.so, libc.so, libdl.so, liblog.so, libm.so, libnativewindow.so, libneuralnetworks.so, libsync.so, libvndksupport.so, and libvulkan.so,
Eligible VNDK Libraries (VNDK) are Framework Shared Libraries that are safe to be copied twice. Framework Modules and Vendor Modules can link with their own copies. A framework shared library can become an eligible VNDK library only if it satisfies the following criteria:
It does not send/receive IPCs to/from the framework.
It is not related to ART virtual machine.
It does not read/write files/partitions with unstable file formats.
It does not have special software license which requires legal reviews.
Its code owner does not have objections to vendor usages.
Framework-Only Libraries (FWK-ONLY) are Framework Shared Libraries that do not belong to the categories mentioned above. These libraries:
Are considered framework internal implementation details.
Must not be accessed by vendor modules.
Have unstable ABIs/APIs and no API/ABI compatibility guarantees.
Are not copied.
Same-Process HAL (SP-HAL)
Same-Process HAL (SP-HAL) is a set of predetermined HALs implemented as Vendor Shared Libraries and loaded into Framework Processes. SP-HALs are isolated by a linker namespace (controls the libraries and symbols that are visible to the shared libraries). SP-HALs must depend only on LL-NDK and VNDK-SP.

VNDK-SP is a predefined subset of eligible VNDK libraries. VNDK-SP libraries are carefully reviewed to ensure double-loading VNDK-SP libraries into framework processes does not cause problems. Both SP-HALs and VNDK-SPs are defined by Google.

The following libraries are approved SP-HALs:

libGLESv1_CM_${driver}.so
libGLESv2_${driver}.so
libGLESv3_${driver}.so
libEGL_${driver}.so
vulkan.${driver}.so
android.hardware.renderscript@1.0-impl.so
android.hardware.graphics.mapper@2.0-impl.so
VNDK-SP libraries specify vndk: { support_system_process: true } in their Android.bp files. If vndk: {private:true} is also specified, then these libraries are called VNDK-SP-Private and they are invisible to SP-HALS.

The following are framework-only libraries with RS exceptions (FWK-ONLY-RS):

libft2.so (Renderscript)
libmediandk.so (Renderscript)

Caution: Starting with Android 12, the RenderScript APIs are deprecated. They will continue to function, but device and component manufacturers are expected to stop providing hardware-acceleration support over time. To take full advantage of GPU acceleration, AOSP recommends that you Migrate from RenderScript.
VNDK versioning
VNDK shared libraries are versioned:

The ro.vndk.version system property is automatically added to /vendor/default.prop.
VNDK and VNDK-SP shared libraries are installed as a VNDK apex com.android.vndk.v${ro.vndk.version} and mounted to /apex/com.android.vndk.v${ro.vndk.version}.
The value of ro.vndk.version is chosen by the algorithm below:

If BOARD_VNDK_VERSION is not equal to current, use BOARD_VNDK_VERSION.
If BOARD_VNDK_VERSION is equal to current:
If PLATFORM_VERSION_CODENAME is REL, use PLATFORM_SDK_VERSION (e.g. 28).
Otherwise, use PLATFORM_VERSION_CODENAME (e.g. P).
Vendor Test Suite (VTS)
The Android Vendor Test Suite (VTS) mandates a non-empty ro.vndk.version property. Both newly launched devices and upgrading devices must define ro.vndk.version. Some VNDK test cases (e.g. VtsVndkFilesTest and VtsVndkDependencyTest) rely on the ro.vndk.version property to load the matching eligible VNDK libraries data sets.

Was this helpful?

Here's a **simple and structured explanation** of the **Vendor Native Development Kit (VNDK)** and why it's **deprecated from Android 15**:

🔧 What is VNDK?

VNDK (Vendor Native Development Kit) is a set of **system shared libraries** that:

- Are used by binaries and libraries in the **vendor** or **product** partition.
- Are designed to allow the **Android framework (system partition)** and **vendor code (vendor partition)** to be updated independently.

🧩 Why Was VNDK Introduced (Android 8.0)?

Before Android 8:

- Vendor and system binaries **directly depended** on each other's libraries.
- This **broke compatibility** when updating only the system (framework) with GSI (Generic System Image).

🔐 Problems It Solved:

1. **Framework-only updates** became possible (i.e., update AOSP framework without touching vendor).
2. Vendor code used **"safe" libraries** only (not internal framework stuff).
3. Avoided errors due to **ABI mismatch** or **double-loading** libraries.

📚 VNDK Library Categories:

1. **LL-NDK (Low-Level NDK):**
   - Stable public APIs (e.g., `libc.so`, `liblog.so`)
   - Safe for both framework and vendor to use.
2. **VNDK Libraries:**
   - Copied into VNDK APEX (one copy for framework, one for vendor)
   - Meet criteria like no IPC, no file format dependence, etc.
3. **Framework-Only Libraries (FWK-ONLY):**
   - Internal implementation details.
   - Must **not** be accessed by vendor.

⚙️ SP-HALs and VNDK-SP:

- **SP-HALs** = HALs loaded into **framework process** (same-process).
- SP-HALs use only **LL-NDK** and **VNDK-SP** libraries.
- VNDK-SP = Carefully selected safe VNDK subset for system use.

📦 VNDK APEX:

- All VNDK libraries are bundled in an **APEX module** (like a container).
- Installed at `/apex/com.android.vndk.v<version>`.

🚫 Why VNDK Was Deprecated (Android 15):

❌ Problems:

1. **Wasted storage**: One big VNDK APEX includes all libraries, even unused ones.
2. **Multiple versions**: GSI must include many VNDK versions for compatibility.
3. **Not easily updatable**: Updating VNDK means updating the whole APEX/system.
4. **Redundant**: Vendor partitions are now OTA-updated regularly.

✅ What Replaces VNDK?

- VNDK libraries are now placed **directly in `/vendor` or `/product`**.
- No APEX needed for them.
- Each vendor build includes only what's actually used.

🔍 What's Removed in Android 15:

- No more `ro.vndk.version`, `ro.product.vndk.version` system properties.
- No `VNDK APEX` for Android 15.
- Build-time flags like `use_vndk_as_stable` and `TARGET_VNDK_USING_CORE_VARIANT` removed.
- Vendor snapshot generation that depended on VNDK is affected.

🔒 What Remains (Not Deprecated):

- VNDK APEX v14 and below **still supported** for legacy devices.
- **LL-NDK** is separate and still used.
- **SP-HALs** and their VNDK-SP dependencies remain.

🧪 VTS and VNDK

- VTS tests like `VtsVndkFilesTest` relied on `ro.vndk.version`.
- Android 15+ devices must **not define this property** anymore if vendor is up-to-date.

🧠 Summary

Feature	Android ≤14	Android 15+
VNDK Libraries	Packaged as APEX in system	Installed in vendor/product
`ro.vndk.version`	Required	Removed
Vendor Snapshot	Uses VNDK info	Changed or removed

Feature	Android ≤14	Android 15+
Updatability	Hard to update	Easier OTA updates
Storage	Large due to all libraries	Smaller (only what's needed)

Let me know if you'd like an illustrated diagram of this flow or a table comparing **VNDK vs APEX-based approach**.

2. Enable VNDK

bookmark_border
The Vendor Native Development Kit (VNDK) requires several changes to a codebase to separate concerns between vendor and system. Use the following guide to enable VNDK in a vendor/OEM codebase.

Build system libraries
The build system contains several types of objects including libraries (shared, static, or header) and binaries.

Build system libraries
Figure 1. Build system libraries.

core libraries are used by the system image, on the system image. These libraries can't be used by vendor, vendor_available, vndk, or vndk-sp libraries.

```
cc_library {
    name: "libThatIsCore",
    ...
}
```
vendor-only (or proprietary) libraries are used by the vendor image, on the vendor image.

```
cc_library {
    name: "libThatIsVendorOnly",
    proprietary: true,
    # or: vendor: true, # (for things in AOSP)
    ...
}
```
vendor_available libraries are used by the vendor image, on the vendor image (may contain duplicates of core).

```
cc_library {
    name: "libThatIsVendorAvailable",
    vendor_available: true,
    ...
}
```
vndk libraries are used by the vendor image, on the system image.

```
cc_library {
    name: "libThatIsVndk",
    vendor_available: true,
    vndk: {
        enabled: true,
    }
    ...
}
```
vndk-sp libraries are used by the vendor image, and also by the system image indirectly.

```
cc_library {
    name: "libThatIsVndkSp",
    vendor_available: true,
    vndk: {
        enabled: true,
        support_system_process: true,
    }
    ...
}
```
llndk libraries are used by both the system and vendor images.

```
cc_library {
    name: "libThatIsLlndk",
    llndk: {
        symbol_file: "libthatisllndk.map.txt"
    }
    ...
```

}
When a lib is marked as vendor_available:true, it's built twice:

Once for platform (and thus installed to /system/lib)
Once for vendor (and thus installed to /vendor/lib or VNDK APEX)
The vendor versions of libs are built with -D__ANDROID_VNDK__. Private system components that may change significantly in future versions of Android are disabled with this flag. In addition, different libraries export a different set of headers (such as liblog). Options specific to a vendor variant of a target can be specified in an Android.bp file in:

target: { vendor: { … } }
Enable VNDK for a codebase
To enable the VNDK for a codebase:

Determine eligibility by calculating the required sizes of vendor.img and system.img partitions.
Enable BOARD_VNDK_VERSION=current. You can add to BoardConfig.mk or build components with it directly (for example, m -j BOARD_VNDK_VERSION=current MY-LIB).
After enabling BOARD_VNDK_VERSION=current, the build system enforces the following dependency and header requirements.

Manage dependencies
A vendor object that depends on a core component that doesn't exist in vndk or as a vendor object must be resolved using one of the following options:

The dependency can be removed.
If the core component is owned by vendor, it can be marked as vendor_available or vendor.
A change making the core object part of the vndk may be upstreamed to Google.
In addition, if a core component has dependencies on a vendor component, the vendor component must be made into a core component or the dependency must be removed in another way (for example, by removing the dependency or by moving the dependency into a vendor component).

Manage headers
Global header dependencies must be removed to enable the build system to know whether to build the headers with or without -D__ANDROID_VNDK__. For example, libutils headers such as utils/StrongPointer.h can still be accessed using the header library libutils_headers.

Some headers (such as unistd.h) can no longer be included transitively but can be included locally.

Finally, the public part of private/android_filesystem_config.h has been moved to cutils/android_filesystem_config.h. To manage these headers, do one of the following:

Remove the dependency to private/android_filesystem_config.h by replacing all AID_* macros with getgrnam/ getpwnam calls if possible. For example:
(uid_t)AID_WIFI becomes getpwnam("wifi")->pw_uid.
(gid_t)AID_SDCARD_R becomes getgrnam("sdcard_r")->gr_gid.
For details, refer to private/android_filesystem_config.h.
For hard-coded AIS, include cutils/android_filesystem_config.h.

Here's a **simple step-by-step explanation** of how to **enable VNDK** in a vendor or OEM codebase, based on the official AOSP guidance you shared:

✅ What Does "Enable VNDK" Mean?

Enabling VNDK means **cleanly separating system (framework) and vendor code** so that:

- Framework (system.img) and vendor (vendor.img) can be **updated independently**.
- Only **allowed shared libraries** are used between them, preventing ABI breakages.

🏗️ 1. Types of Libraries in Android Build System

Each library must declare **where it's used** (system/vendor) and whether it's part of VNDK.

Library Type	Used From	Declared in `Android.bp` with:
core	system only	*No vendor access*
vendor	vendor only	`vendor: true` or `proprietary: true`

Library Type	Used From	Declared in `Android.bp` with:
vendor_available	vendor only	`vendor_available: true`
vndk	vendor use, lives in system	`vendor_available: true`, `vndk: { enabled: true }`
vndk-sp	vendor + system (SP-HAL)	`vndk: { enabled: true, support_system_process: true }`
llndk	stable vendor+system	`llndk: { symbol_file: "lib.map.txt" }`

📌 **Tip:** `vendor_available:true` makes it build **twice**:

- Once for system (goes to `/system/lib`)
- Once for vendor (goes to `/vendor/lib`)

🧠 2. Enable VNDK for the Codebase

To activate VNDK support, follow these steps:

✅ Step 1: Add this in `BoardConfig.mk`

```make
BOARD_VNDK_VERSION := current
```

This tells the build system to **enforce VNDK rules** and version the libraries.

✅ Step 2: Ensure vendor partition has space

- Make sure the `vendor.img` and `system.img` have enough size.
- Because libraries are now **split**, storage sizes may change.

🔗 3. Manage Dependencies Between System & Vendor

🛑 You cannot do this:

- Vendor binary depending on system (core) library not in VNDK.
- System binary depending on vendor library.

✅ Solutions:

- If vendor depends on a system library:
  - Remove the dependency.
  - Or mark the library as `vendor_available: true`
  - Or upstream it to be included in `vndk`.
- If system depends on vendor code:
  - Move the vendor library to system.
  - Or eliminate the dependency.

🧾 4. Handle Header Files

Problem:

Some headers like `StrongPointer.h`, `unistd.h`, or `android_filesystem_config.h` are **not allowed globally anymore**.

✅ What to Do:

- Use proper **header-only libraries**:

```bp
cc_library_headers {
    name: "libutils_headers",
```

```
    export_include_dirs: ["include"]
}
```

- Avoid using private headers like:

```arduino
private/android_filesystem_config.h
```

- Use:
  - **`cutils/android_filesystem_config.h`** for hardcoded AIDs.
  - Or dynamically fetch UID/GID using:

    ```cpp
    getpwnam("wifi")->pw_uid;
    getgrnam("sdcard_r")->gr_gid;
    ```

🎯 Summary

Task	What to Do
Enable VNDK	**`BOARD_VNDK_VERSION := current`**
Mark eligible libs	Use **`vndk`**, **`vendor_available`**, **`llndk`** correctly
Split builds	Build vendor variants separately
Fix headers	Replace global/private headers with clean ones
Fix dependencies	Break illegal system↔vendor dependencies

Would you like:

- 🔍 A sample **`Android.bp`** file with all these flags?
- 🗂️ Folder layout for a vendor module with VNDK?
- 📊 Diagram showing how these libraries are split at runtime?

Let me know and I can provide!

3. ⤷
`llndk`
what is this

Great question!

🔍 What is `llndk` in Android?

`**llndk**` stands for **Low-Level Native Development Kit**.

It is a **set of system shared libraries** that are:

- **Stable and public**
- **Used by both system (framework) and vendor code**
- **Maintained by Google with strict ABI/API stability**

🔑 Purpose of `llndk`:

- To **allow vendor code** to safely use certain **core Android system libraries** without risk of breaking in future updates.
- To **avoid duplication** of common libraries in both system and vendor partitions.
- To ensure **compatibility with GSI** (Generic System Image) for vendor testing.

🛠️ How to define an `llndk` library in Android.bp:

```bp
cc_library {
    name: "libmyllndk",
    llndk: {
        symbol_file: "libmyllndk.map.txt",
    },
    ...
}
```

- `symbol_file` contains the list of allowed public symbols (functions, variables).
- These symbols are exported **only** and **guaranteed to stay stable** across Android versions.

📦 Examples of `llndk` Libraries:

Some commonly used `llndk` libraries include:

- `libc.so` — standard C library
- `liblog.so` — logging support
- `libdl.so` — dynamic linker interface
- `libm.so` — math functions
- `libsync.so`, `libnativewindow.so`, `libvndksupport.so`, etc.

These are all **safe for vendor HALs** to use.

🧠 Summary Table:

Feature	`llndk`
Scope	System + Vendor
Stability	✅ Stable across versions

Feature	`llndk`
Location	**`/system/lib[64]`**
Symbol control	Via **`.map.txt`** file
ABI Compatibility	Strictly maintained
Use case	HALs, vendor libs needing system API

Would you like to see:

- A sample **`libllndk.map.txt`** file?
- How **`llndk`** differs from **`vndk`** and **`vndk-sp`**?

Let me know!

4. VNDK build system support

bookmark_border
In Android 8.1 and higher, the build system has built-in VNDK support. When VNDK support is enabled, the build system checks the dependencies between modules, builds a vendor-specific variant for vendor modules, and automatically installs those modules into designated directories.

VNDK build support example
In this example, the Android.bp module definition defines a library named libexample. The vendor_available property indicates framework modules and vendor modules may depend on libexample:

libexample vendor_available:true and vndk.enabled:true

Figure 1. support enabled.

Both the framework executable /system/bin/foo and the vendor executable /vendor/bin/bar depend on libexample and have libexample in their shared_libs properties.

If libexample is used by both framework modules and vendor modules, two variants of libexample are built. The core variant (named after libexample) is used by framework modules and the vendor variant (named after libexample.vendor) is used by vendor modules. The two variants are installed into different directories:

The core variant is installed into /system/lib[64]/libexample.so.
The vendor variant is installed into VNDK APEX because vndk.enabled is true.
For more details, see Module definition.

Configure build support
To enable full build system support for a product device, add BOARD_VNDK_VERSION to BoardConfig.mk:

BOARD_VNDK_VERSION := current
This setting has a global effect: When defined in BoardConfig.mk, all modules are checked. As there is no mechanism to blacklist or whitelist an offending module, you should clean all unnecessary dependencies before adding BOARD_VNDK_VERSION. You can test and compile a module by setting BOARD_VNDK_VERSION in your environment variables:

$ BOARD_VNDK_VERSION=current m module_name.vendor
When BOARD_VNDK_VERSION is enabled, several default global header search paths are removed. These include:

frameworks/av/include
frameworks/native/include
frameworks/native/opengl/include
hardware/libhardware/include
hardware/libhardware_legacy/include
hardware/ril/include
libnativehelper/include
libnativehelper/include_deprecated
system/core/include
system/media/audio/include
If a module depends on the headers from these directories, you must specify (explicitly) the dependencies with header_libs, static_libs, and/or shared_libs.

VNDK APEX
In Android 10 and lower, modules with vndk.enabled were installed in /system/lib[64]/vndk[-sp]-${VER}. In Android 11 and higher, VNDK libraries are packaged in an APEX format and the name of VNDK APEX is com.android.vndk.v${VER}. Depending on the device configuration, VNDK APEX is flattened or unflattened and is available from the canonical path /apex/com.android.vndk.v${VER}.

VNDK APEX

Figure 2. VNDK APEX.

Module definition
To build Android with BOARD_VNDK_VERSION, you must revise the module definition in either Android.mk or Android.bp. This section describes different kinds of module definitions, several VNDK-related module properties, and dependency checks implemented in the build system.

Vendor modules
Vendor modules are vendor-specific executables or shared libraries that must be installed into a vendor partition. In Android.bp files, vendor modules must set vendor or proprietary property to true. In Android.mk files, vendor modules must set LOCAL_VENDOR_MODULE or LOCAL_PROPRIETARY_MODULE to true.

If BOARD_VNDK_VERSION is defined, the build system disallows dependencies between vendor modules and framework modules and emits errors if:

a module without vendor:true depends on a module with vendor:true, or
a module with vendor:true depends on a non-llndk_library module that has neither vendor:true nor vendor_available:true.
The dependency check applies to header_libs, static_libs, and shared_libs in Android.bp, and to LOCAL_HEADER_LIBRARIES, LOCAL_STATIC_LIBRARIES and LOCAL_SHARED_LIBRARIES in Android.mk.

LL-NDK
LL-NDK shared libraries are shared libraries with stable ABIs. Both framework and vendor modules share the same and the latest implementation. For each LL-NDK shared library, the cc_library contains a llndk property with a symbol file:

```
cc_library {
  name: "libvndksupport",
  llndk: {
    symbol_file: "libvndksupport.map.txt",
  },
}
```

The symbol file describes the symbols visible to vendor modules. For example:

```
LIBVNDKSUPPORT {
 global:
  android_load_sphal_library; # llndk
  android_unload_sphal_library; # llndk
 local:
  *;
};
```

Based on the symbol file, the build system generates a stub shared library for vendor modules, which link with these libraries when BOARD_VNDK_VERSION is enabled. A symbol is included in the stub shared library only if it:

Is not defined in the section end with _PRIVATE or _PLATFORM,
Does not have #platform-only tag, and
Does not have #introduce* tags or the tag matches with the target.
Note: Vendors must not define their own LL-NDK shared libraries because vendor modules won't be able to find them in Generic System Image (GSI).
VNDK
In Android.bp files, cc_library, cc_library_static, cc_library_shared, and cc_library_headers module definitions support three VNDK-related properties: vendor_available, vndk.enabled, and vndk.support_system_process.

If vendor_available or vndk.enabled is true, two variants (core and vendor) may be built. The core variant should be treated as a framework module and the vendor variant should be treated as a vendor module. If some framework modules depend on this module, the core variant is built. If some vendor modules depend on this module, the vendor variant is built. The build system enforces the following dependency checks:

The core variant is always framework-only and inaccessible to vendor modules.
The vendor variant is always inaccessible to framework modules.
All dependencies of the vendor variant, which are specified in header_libs, static_libs, and/or shared_libs, must be either an llndk_library or a module with vendor_available or vndk.enabled.
If vendor_available is true, the vendor variant is accessible to all vendor modules.
If vendor_available is false, the vendor variant is accessible only to other VNDK or VNDK-SP modules (i.e., modules with vendor:true cannot link vendor_available:false modules).
The default installation path for cc_library or cc_library_shared is determined by the following rules:

The core variant is installed to /system/lib[64].
The vendor variant installation path may vary:
If vndk.enabled is false, the vendor variant is installed into /vendor/lib[64].
If vndk.enabled is true, the vendor variant is installed into VNDK APEX(com.android.vndk.v${VER}).
The table below summarizes how the build system handles the vendor variants:

vendor_available vndk
enabled vndk
support_system_process Vendor variant descriptions
true false false The vendor variants are VND-ONLY. Shared libraries are installed into /vendor/lib[64].
true Invalid (Build error)
true false The vendor variants are VNDK. Shared libraries are installed to VNDK APEX.
true The vendor variants are VNDK-SP. Shared libraries are installed to VNDK APEX.
false

false

false

No vendor variants. This module is FWK-ONLY.

true Invalid (Build error)
true false The vendor variants are VNDK-Private. Shared libraries are installed to VNDK APEX. These must not be directly used by vendor modules.
true The vendor variants are VNDK-SP-Private. Shared libraries are installed to VNDK APEX. These must not be directly used by vendor modules.
Note: Vendors may set vendor_available to their modules but must not set vndk.enabled or vndk.support_system_process because the modules won't be able to find them in the Generic System Image (GSI).
VNDK extensions
VNDK extensions are VNDK shared libraries with additional APIs. Extensions are installed to /vendor/lib[64]/vndk[-sp] (without version suffix) and override the original VNDK shared libraries at runtime.

Define VNDK extensions
In Android 9 and higher, Android.bp natively supports VNDK extensions. To build a VNDK extension, define another module with a vendor:true and an extends property:

```
cc_library {
    name: "libvndk",
    vendor_available: true,
    vndk: {
        enabled: true,
    },
}

cc_library {
    name: "libvndk_ext",
    vendor: true,
    vndk: {
        enabled: true,
        extends: "libvndk",
    },
}
```

A module with vendor:true, vndk.enabled:true, and extends properties defines the VNDK extension:

The extends property must specify a base VNDK shared library name (or VNDK-SP shared library name).
VNDK extensions (or VNDK-SP extensions) are named after the base module names from which they extend. For example, the output binary of libvndk_ext is libvndk.so instead of libvndk_ext.so.
VNDK extensions are installed into /vendor/lib[64]/vndk.
VNDK-SP extensions are installed into /vendor/lib[64]/vndk-sp.
The base shared libraries must have both vndk.enabled:true and vendor_available:true.
A VNDK-SP extension must extend from a VNDK-SP shared library (vndk.support_system_process must be equal):

```
cc_library {
    name: "libvndk_sp",
    vendor_available: true,
    vndk: {
        enabled: true,
        support_system_process: true,
    },
}
```

```
cc_library {
    name: "libvndk_sp_ext",
    vendor: true,
    vndk: {
        enabled: true,
        extends: "libvndk_sp",
        support_system_process: true,
    },
}
```

VNDK extensions (or VNDK-SP extensions) may depend on other vendor shared libraries:

```
cc_library {
    name: "libvndk",
    vendor_available: true,
    vndk: {
        enabled: true,
    },
}

cc_library {
    name: "libvndk_ext",
    vendor: true,
    vndk: {
        enabled: true,
        extends: "libvndk",
    },
    shared_libs: [
        "libvendor",
    ],
}

cc_library {
    name: "libvendor",
    vendor: true,
}
```

Note: Similar to SP-HAL-Dep, VNDK-SP extensions and their dependencies (including vendor libraries) must be labeled as same_process_hal_file in sepolicy.
Use VNDK extensions
If a vendor module depends on additional APIs defined by VNDK extensions, the module must specify the name of the VNDK extension in its shared_libs property:

```
// A vendor shared library example
cc_library {
    name: "libvendor",
    vendor: true,
    shared_libs: [
        "libvndk_ext",
    ],
}

// A vendor executable example
cc_binary {
    name: "vendor-example",
    vendor: true,
    shared_libs: [
        "libvndk_ext",
    ],
}
```

If a vendor module depends on VNDK extensions, those VNDK extensions are installed to /vendor/lib[64]/vndk[-sp] automatically. If a module no longer depends on a VNDK extension, add a clean step to CleanSpec.mk to remove the shared library. For example:

```
$(call add-clean-step, rm -rf $(TARGET_OUT_VENDOR)/lib/libvndk.so)
```

Conditional compilation
This section describes how to deal with the subtle differences (e.g. adding or removing a feature from one of the variants) between the following three VNDK shared libraries:

Core variant (e.g. /system/lib[64]/libexample.so)
Vendor variant (e.g. /apex/com.android.vndk.v${VER}/lib[64]/libexample.so)
VNDK extension (e.g. /vendor/lib[64]/vndk[-sp]/libexample.so)
Conditional compiler flags
The Android build system defines __ANDROID_VNDK__ for vendor variants and VNDK extensions by default. You may guard the code with the C preprocessor guards:

```
void all() { }

#if !defined(__ANDROID_VNDK__)
void framework_only() { }
#endif

#if defined(__ANDROID_VNDK__)
void vndk_only() { }
#endif
```

In addition to __ANDROID_VNDK__, different cflags or cppflags may be specified in Android.bp. The cflags or cppflags specified in target.vendor is specific to the vendor variant.

For example, the following Android.bp defines libexample and libexample_ext:

```
cc_library {
    name: "libexample",
    srcs: ["src/example.c"],
    vendor_available: true,
    vndk: {
        enabled: true,
    },
    target: {
        vendor: {
            cflags: ["-DLIBEXAMPLE_ENABLE_VNDK=1"],
        },
    },
}

cc_library {
    name: "libexample_ext",
    srcs: ["src/example.c"],
    vendor: true,
    vndk: {
        enabled: true,
        extends: "libexample",
    },
    cflags: [
        "-DLIBEXAMPLE_ENABLE_VNDK=1",
        "-DLIBEXAMPLE_ENABLE_VNDK_EXT=1",
    ],
}
```

And this is the code listing of src/example.c:

```
void all() { }

#if !defined(LIBEXAMPLE_ENABLE_VNDK)
void framework_only() { }
#endif

#if defined(LIBEXAMPLE_ENABLE_VNDK)
void vndk() { }
#endif
```

```
#if defined(LIBEXAMPLE_ENABLE_VNDK_EXT)
void vndk_ext() { }
#endif
```

According to these two files, the build system generates shared libraries with following exported symbols:

Installation path Exported symbols
/system/lib[64]/libexample.so all, framework_only
/apex/com.android.vndk.v${VER}/lib[64]/libexample.so all, vndk
/vendor/lib[64]/vndk/libexample.so all, vndk, vndk_ext

Requirements on the exported symbols
The VNDK ABI checker compares the ABI of VNDK vendor variants and VNDK extensions to the reference ABI dumps under prebuilts/abi-dumps/vndk.

Symbols exported by VNDK vendor variants (e.g. /apex/com.android.vndk.v${VER}/lib[64]/libexample.so) must be identical to (not the supersets of) the symbols defined in ABI dumps.
Symbols exported by VNDK extensions (e.g. /vendor/lib[64]/vndk/libexample.so) must be supersets of the symbols defined in ABI dumps.
If VNDK vendor variants or VNDK extensions fail to follow the requirements above, VNDK ABI checker emits build errors and stops the build.

Exclude source files or shared libraries from vendor variants
To exclude source files from the vendor variant, add them to the exclude_srcs property. Similarly, to ensure shared libraries are not linked with the vendor variant, add those libraries to the exclude_shared_libs property. For example:

```
cc_library {
    name: "libexample_cond_exclude",
    srcs: ["fwk.c", "both.c"],
    shared_libs: ["libfwk_only", "libboth"],
    vendor_available: true,
    target: {
        vendor: {
            exclude_srcs: ["fwk.c"],
            exclude_shared_libs: ["libfwk_only"],
        },
    },
}
```

In this example, the core variant of libexample_cond_exclude includes the code from fwk.c and both.c and depends on the shared libraries libfwk_only and libboth. The vendor variant of libexample_cond_exclude includes only the code from both.c because fwk.c is excluded by the exclude_srcs property. Similarly, it depends on only the shared library libboth because libfwk_only is excluded by the exclude_shared_libs property.

Export headers from VNDK extensions
A VNDK extension may add new classes or new functions to a VNDK shared library. It is suggested to keep those declarations in independent headers and avoid changing the existing headers.

For example, a new header file include-ext/example/ext/feature_name.h is created for the VNDK extension libexample_ext:

Android.bp
include-ext/example/ext/feature_name.h
include/example/example.h
src/example.c
src/ext/feature_name.c

In the following Android.bp, libexample exports only include, whereas libexample_ext exports both include and include-ext. This ensures feature_name.h won't be incorrectly included by the users of libexample:

```
cc_library {
    name: "libexample",
    srcs: ["src/example.c"],
    export_include_dirs: ["include"],
    vendor_available: true,
    vndk: {
        enabled: true,
    },
}

cc_library {
```

```
  name: "libexample_ext",
  srcs: [
    "src/example.c",
    "src/ext/feature_name.c",
  ],
  export_include_dirs: [
    "include",
    "include-ext",
  ],
  vendor: true,
  vndk: {
    enabled: true,
    extends: "libexample",
  },
}
```

If separating extensions to independent header files is not feasible, an alternative is to add #ifdef guards. However, make sure that all VNDK extension users add the define flags. You may define cc_defaults to add define flags to cflags and link shared libraries with shared_libs.

For example, to add a new member function Example2::get_b() to the VNDK extension libexample2_ext, you must modify the existing header file and add a #ifdef guard:

```
#ifndef LIBEXAMPLE2_EXAMPLE_H_
#define LIBEXAMPLE2_EXAMPLE_H_

class Example2 {
 public:
  Example2();

  void get_a();

#ifdef LIBEXAMPLE2_ENABLE_VNDK_EXT
  void get_b();
#endif

 private:
  void *impl_;
};

#endif  // LIBEXAMPLE2_EXAMPLE_H_
```

A cc_defaults named libexample2_ext_defaults is defined for the users of libexample2_ext:

```
cc_library {
  name: "libexample2",
  srcs: ["src/example2.cpp"],
  export_include_dirs: ["include"],
  vendor_available: true,
  vndk: {
    enabled: true,
  },
}

cc_library {
  name: "libexample2_ext",
  srcs: ["src/example2.cpp"],
  export_include_dirs: ["include"],
  vendor: true,
  vndk: {
    enabled: true,
    extends: "libexample2",
  },
  cflags: [
    "-DLIBEXAMPLE2_ENABLE_VNDK_EXT=1",
  ],
}
```

```
cc_defaults {
    name: "libexample2_ext_defaults",
    shared_libs: [
        "libexample2_ext",
    ],
    cflags: [
        "-DLIBEXAMPLE2_ENABLE_VNDK_EXT=1",
    ],
}
The users of libexample2_ext may simply include libexample2_ext_defaults in their defaults property:


cc_binary {
    name: "example2_user_executable",
    defaults: ["libexample2_ext_defaults"],
    vendor: true,
}
Product packages
In the Android build system, the variable PRODUCT_PACKAGES specifies the executables, shared libraries, or packages that should be installed
into the device. The transitive dependencies of the specified modules are implicitly installed into the device as well.

If BOARD_VNDK_VERSION is enabled, modules with vendor_available or vndk.enabled get special treatment. If a framework module depends on a
module with vendor_available or vndk.enabled, the core variant is included in the transitive installation set. If a vendor module depends on a
module with vendor_available, the vendor variant is included in the transitive installation set. However, vendor variants of modules with
vndk.enabled are installed whether or not they are used by vendor modules.

When the dependencies are invisible to the build system (e.g. shared libraries that may be opened with dlopen() in runtime), you should specify
the module names in PRODUCT_PACKAGES to install those modules explicitly.

If a module has vendor_available or vndk.enabled, the module name stands for its core variant. To explicitly specify the vendor variant in
PRODUCT_PACKAGES, append a .vendor suffix to the module name. For example:


cc_library {
    name: "libexample",
    srcs: ["example.c"],
    vendor_available: true,
}
In this example, libexample stands for /system/lib[64]/libexample.so and libexample.vendor stands for /vendor/lib[64]/libexample.so. To install
/vendor/lib[64]/libexample.so, add libexample.vendor to PRODUCT_PACKAGES:


PRODUCT_PACKAGES += libexample.vendor
```

You're looking at a **deep-dive into VNDK (Vendor Native Development Kit) build system support** in Android. Here's a **clear breakdown** to understand what it means, what it does, and how it works—**in simple terms**:

🧠 What Is This Document About?

It explains:

- How the **Android build system supports VNDK** (starting Android 8.1+)
- What changes are required in `Android.bp` or `Android.mk`
- How to **build vendor-safe libraries**
- How to handle **VNDK extensions**, **VNDK APEX**, and **conditional compilation**
- How libraries are **split between system and vendor**

🔧 Key Concepts Explained Simply

🔹 What Happens When You Enable VNDK?

You write in **`BoardConfig.mk`**:

```make
BOARD_VNDK_VERSION := current
```

Then:

1. Build system checks all module dependencies.
2. Some libraries will be built **twice**:
   - A **core (framework)** version → goes to **`/system/lib`**
   - A **vendor** version → goes to **`/vendor/lib`** or **`/apex/...`**
3. These are split so that **vendor code doesn't rely on unstable framework internals**.

📁 Library Variant Split

When both framework and vendor modules use the same library, like **`libexample`**:

```bp
cc_library {
    name: "libexample",
    vendor_available: true,
    vndk: {
        enabled: true,
    }
}
```

You get:

Variant	Installed To
Core	**`/system/lib[64]/libexample.so`**
Vendor	**`/apex/com.android.vndk.v${VER}/lib[64]/libexample.so`**

📦 VNDK APEX

Starting from Android 11:

- All VNDK libraries are bundled into an **APEX module** named like:

  ```
  com.android.vndk.v34
  ```

- Mounted to:

  ```bash
  /apex/com.android.vndk.v34
  ```

These libraries are read-only and versioned.

🧩 Types of Modules

Module Type	Flags in Android.bp	Lives In
Vendor-only	**`vendor: true`** or **`proprietary: true`**	**`/vendor`**
Vendor Available	**`vendor_available: true`**	Both vendor & system
VNDK	**`vndk.enabled: true`**	APEX
VNDK-SP	**`support_system_process: true`**	APEX
LLNDK	**`llndk.symbol_file: ...`**	Shared with vendor

🔁 Dependency Rules (very important)

- Vendor code **must not link** to framework-only modules.
- A vendor module can only depend on:
  - `vendor_available` libraries
  - `llndk` libraries
  - VNDK libraries

🌱 VNDK Extensions

If vendor wants to **extend a VNDK library** (e.g. add more APIs), they can override it via:

```bp
cc_library {
    name: "libvndk_ext",
    vendor: true,
    vndk: {
        enabled: true,
        extends: "libvndk",
    }
}
```

This gets installed to:

```bash
/vendor/lib[64]/vndk/
```

And it **overrides** the system version **only for vendor code**.

💡 Conditional Compilation

You can write code that behaves differently depending on whether it's being built for system or vendor:

```cpp
#if defined(__ANDROID_VNDK__)
void only_for_vendor_builds() {}
#else
void only_for_framework() {}
#endif
```

Build system **automatically defines** `__ANDROID_VNDK__` for vendor builds.

📦 Installing Vendor Variant in PRODUCT_PACKAGES

If you need the vendor version of a library (e.g. for `dlopen()` at runtime), add it like:

```make
PRODUCT_PACKAGES += libexample.vendor
```

This ensures `/vendor/lib[64]/libexample.so` is actually included in the final image.

📘 Summary Table

Feature	Purpose
`vendor_available: true`	Make a module accessible to vendor
`vndk.enabled: true`	Build vendor variant and include in VNDK APEX
`llndk.symbol_file`	Define safe symbols for both vendor and system
`extends: "libfoo"`	Create a VNDK extension overriding `libfoo`

Feature	Purpose
`exclude_srcs`, **`exclude_shared_libs`**	Exclude code/libs from vendor variant
`__ANDROID_VNDK__`	Guard code paths for vendor/system differences
`.vendor` suffix in PRODUCT_PACKAGES	Ensure vendor variant is included

---

Would you like:

- ✅ A visual diagram showing how system vs vendor library variants are split?
- ✅ A full **`Android.bp`** + code example using **`vndk.enabled`**, **`extends`**, and conditional flags?

Just say the word and I'll provide it inline.

5. ↳
📦 **Installing Vendor Variant in PRODUCT_PACKAGES** -- If you need the vendor version of a library (e.g. for `dlopen()` at runtime), add it like: make CopyEdit `PRODUCT_PACKAGES += libexample.vendor` This ensures `/vendor/lib[64]/libexample.so` is actually included in the final image.
explain this

Great — let's explain this part **step by step in simple words**:

✅ What is `PRODUCT_PACKAGES`?

In Android's build system (**`Android.mk`** or **`Android.bp`**), there's a variable called **`PRODUCT_PACKAGES`**.

You use it to **tell the build system**:

> "Hey, please install this module (app, binary, shared library, etc.) into the final system image."

🧠 What's the Issue with VNDK or Vendor Libraries?

When a module (like a shared library) is marked with:

```bp
vendor_available: true
```

or

```bp
vndk: {
    enabled: true
}
```

...the **build system builds two versions** of it:

Variant	Path	Use case
Core	**`/system/lib[64]/libexample.so`**	Used by framework
Vendor	**`/vendor/lib[64]/libexample.so`** (or VNDK APEX)	Used by vendor binaries

✅ This is good — vendor and framework don't step on each other.

🚫 But Here's the Problem

If a vendor binary wants to **dynamically load** the library at runtime using:

```c
dlopen("libexample.so", RTLD_NOW);
```

...it expects the **`.so`** file to **exist at runtime** in the vendor partition.

🧨 However, if you didn't explicitly tell the build system to include **`libexample.vendor`** in your final image, it may **not install it**, especially if it's only loaded via **`dlopen()`** and not linked statically.

💡 Solution: Add `.vendor` in `PRODUCT_PACKAGES`

So you do this in your device **`.mk`** file (like **`device.mk`** or **`product.mk`**):

```make
PRODUCT_PACKAGES += libexample.vendor
```

This tells the build system:

> "Install the **vendor variant** of `libexample` to `/vendor/lib[64]/libexample.so`."

🔁 Summary

Situation	What you should do
Vendor binary needs `libexample.so` at runtime via `dlopen()`	Add `libexample.vendor` to `PRODUCT_PACKAGES`
You don't add it	Library might not be present at runtime = crash

---

🧪 Example

If you have this:

```bp
cc_library {
    name: "libexample",
    srcs: ["example.c"],
    vendor_available: true,
}
```

And you're writing a vendor binary:

```bp
cc_binary {
    name: "my_hal_service",
    srcs: ["main.c"],
    vendor: true,
    shared_libs: ["libexample"],  // or loaded via dlopen()
}
```

Add this to your device `.mk`:

```make
PRODUCT_PACKAGES += libexample.vendor
```

So that the **vendor version** is installed into your `/vendor/lib[64]` folder during build.

Let me know if you want a real working example with `Android.bp` + `device.mk` + sample `dlopen()` code!

6. VNDK extensions

bookmark_border
Android device manufacturers change the source code of AOSP libraries for various reasons. Some vendors reimplement functions in AOSP libraries to boost the performance while other vendors add new hooks, new APIs, or new functionalities to AOSP libraries. This section provides guidelines for extending AOSP libraries in a way that does not break CTS/VTS.

Drop-in replacement
All modified shared libraries must be binary-compatible, drop-in replacements of their AOSP counterpart. All existing AOSP users must be able to use the modified shared library without recompilations. This requirement implies the following:

AOSP functions must not be removed.
Structures must not be altered if such structures are exposed to their users.
Pre-condition of functions must not be strengthened.
Functions must provide equivalent functionalities.
Post-condition of functions must not be weakened.
Extended module classifications
Classify modules by the functionalities they define and use.

Note: Functionalities is used here instead of API/ABI because it is possible to add functionality without changing any API/ABI.

Depending on the functionalities defined in a module, modules can be classified into DA-Module and DX-Module:

Defining-only-AOSP Modules (DA-Module) do not define new functionalities which were not in the AOSP counterpart.
Example 1. An intact unmodified AOSP library is a DA-Module.
Example 2. If a vendor rewrites the functions in libcrypto.so with SIMD instructions (without adding new functions), then the modified libcrypto.so will be a DA-Module.
Defining-Extension Modules (DX-Module) either define new functionalities or do not have an AOSP counterpart.
Example 1. If a vendor adds a helper function to libjpeg.so to access some internal data, then the modified libjpeg.so will be a DX-Lib and the newly added function will be the extended portion of the library.
Example 2. If a vendor defines a non-AOSP library named libfoo.so, then libfoo.so will be a DX-Lib.
Depending on the functionalities used by a module, modules can be classified into UA-Module and UX-Module.

Using-only-AOSP Modules (UA-Module) only use AOSP functionalities in their implementations. They do not rely on any non-AOSP extensions.
Example 1. An intact unmodified AOSP library is an UA-Module.
Example 2. If a modified shared library libjpeg.so only relies on other AOSP APIs, then it will be an UA-Module.
Using-Extension Modules (UX-Module) rely on some non-AOSP functionalities in their implementations.
Example 1. If a modified libjpeg.so relies on another non-AOSP library named libjpeg_turbo2.so, then the modified libjpeg.so will be an UX-Module.
Example 2. If a vendor adds a new function to their modified libexif.so and their modified libjpeg.so uses the newly added function from libexif.so, then their modified libjpeg.so will be an UX-Module.
Definitions and usages are independent from each other:

Used Functionalities
Only AOSP (UA)	Extended (UX)
Defined Functionalities	Only AOSP (DA)	DAUA	DAUX
Extended (DX)	DXUA	DXUX
VNDK extension mechanism
Vendor modules that rely on extended functionalities won't work because the AOSP library with the same name does not have the extended functionality. If vendor modules directly or indirectly depend on extended functionalities, vendors should copy DAUX, DXUA, and DXUX shared libraries to the vendor partition (vendor processes always look for shared libraries in the vendor partition first). However, LL-NDK libraries must not be copied, so vendor modules must not rely on the extended functionalities defined by the modified LL-NDK libraries.

DAUA shared libraries can remain on the system partition if the corresponding AOSP library can provide the same functionality and vendor modules continue to work when the system partition is overwritten by a Generic System Image (GSI).

Drop-in replacement is important because the unmodified VNDK libraries in the GSI will link with the modified shared libraries on name collision. If the AOSP libraries are modified in an API/ABI incompatible manner, the AOSP libraries in the GSI might fail to link or result in undefined behaviors.

This section explains **how vendors (OEMs/ODMs)** can safely **extend AOSP libraries** (i.e., shared `.so` libraries) without breaking **GSI compatibility**, **VNDK rules**, or failing **CTS/VTS tests**.

Let me explain all of it **step-by-step in simple terms**:

✅ Why do vendors modify AOSP libraries?

Vendors often modify AOSP `.so` libraries for:

- Better **performance** (e.g., SIMD-optimized `libcrypto.so`)
- Adding **new APIs** or **hooks** (e.g., helper functions)
- Adding device-specific **features**

BUT—these modifications can **break compatibility** with GSI (Generic System Image) or CTS (Compatibility Test Suite) if done carelessly.

🚨 The Golden Rule: Must be a drop-in replacement

This means:

- Your modified `.so` must be **binary compatible** with the AOSP version
- It must work **without requiring recompilation** of other system components

That means:

Rule	Explanation
❌ Don't remove AOSP functions	All functions must stay
❌ Don't change public structs	They must remain same layout
❌ Don't make function behavior stricter	e.g., function should still accept same inputs
✅ Do provide the same output and contract	Preserve behavior and ABI

📦 Classification of Modified Libraries

To help manage extensions, Android classifies libraries in two **dimensions**:

1. What the module defines (exports):

Term	Meaning	Example
DA-Module (Defines AOSP-only functions)	You rewrote the code, but didn't add new functions	SIMD-optimized `libcrypto.so`
DX-Module (Defines Extensions)	You added new functions	Modified `libjpeg.so` with new helper

2. What the module uses (depends on):

Term	Meaning	Example
UA-Module (Uses AOSP only)	Only uses existing AOSP functionality	Modified `libjpeg.so` using only AOSP headers
UX-Module (Uses Extension)	Uses other vendor-added stuff	Modified `libjpeg.so` calling new API in vendor `libexif.so`

💡 Combine them into 4 types:

Type	Meaning
DAUA	Only defines and uses AOSP stuff
DAUX	Only defines AOSP stuff, but uses vendor extension
DXUA	Defines new APIs, but only uses AOSP underneath
DXUX	Defines new APIs and also uses vendor stuff

🧩 Problem with Vendor Extensions

Vendor processes load libraries from `/vendor/lib` before `/system/lib`.

That means if you:

- Extend AOSP libraries improperly,
- Or modify **LL-NDK** libraries (which are meant to be shared between vendor and system),

You can **break compatibility** with GSI or system apps.

🛠️ Solution — Use VNDK Extensions

✅ Allowed:

You may install your modified `.so` (DAUX/DXUA/DXUX) to:

```bash
/vendor/lib[64]/vndk[-sp]
```

And override AOSP versions **safely** at runtime.

❌ Not allowed:

You **cannot** override **LL-NDK** libraries (like `libc.so`, `liblog.so`, `libnativewindow.so`) because:

- They're shared by system and vendor
- Modifying them breaks the contract

🧪 Example:

Scenario:

You modify `libjpeg.so` to add `void get_internal_jpeg_state();`

Now:

- You're defining new functionality → **DX**
- Your code calls a new helper in `libexif_vendor.so` → **UX**

→ This becomes a **DXUX** module.

📦 To support this, put your new `libjpeg.so` into:

```bash
/vendor/lib[64]/vndk/libjpeg.so
```

Any vendor binary that needs it will link against this version instead of the system one.

⚠️ Why Drop-in Compatibility Matters

When you **boot with a GSI** (Generic System Image), it brings in **unmodified AOSP `.so` files**.

If your vendor `.so`:

- **removes symbols**
- **changes struct sizes**
- or **breaks binary compatibility**

...the GSI **will crash** due to undefined behavior or linking failures.

So: **preserve AOSP ABI** → never break it.

🧵 Summary

Do	Don’t
✅ Use VNDK Extensions for custom APIs	❌ Modify LL-NDK libraries
✅ Preserve AOSP ABI and binary behavior	❌ Remove or change AOSP symbols
✅ Classify modules (DA, DX, UA, UX)	❌ Mix incompatible APIs across system/vendor
✅ Install DX/UX extensions under **`/vendor/lib[64]/vndk`**	❌ Install incompatible **`.so`** to system/lib

---

Would you like a **visual diagram** of this DAUA/DXUX matrix and how loading works in vendor vs system process?

7. can you explain these with example

Yes, let's go through **clear examples** for each concept related to **VNDK extensions** using **realistic AOSP-style libraries**. I'll cover:

🔰 First, assume:

You are a **vendor** working on a device with these libraries:

- `libjpeg.so`: an AOSP library used for image decoding
- `libexif.so`: another AOSP library for metadata
- `libvendorcodec.so`: a custom library added by the vendor

📦 EXAMPLES OF VNDK EXTENSION MODULE TYPES

🟩 1. DAUA (Defines AOSP only, Uses AOSP only)

Scenario: You optimized `libcrypto.so` using SIMD, but didn't add any new APIs.

```c
// Original AOSP function
int encrypt(const char* input, char* output) {
    // AOSP implementation
}
```

You replace the body with optimized code, but keep the same function signature.

🔧 **Result**:

- Defines only AOSP APIs → **DA**
- Uses only AOSP internals → **UA**
  ✅ Safe replacement.

📁 **Installed in**:

```bash
/vendor/lib[64]/vndk/libcrypto.so
```

🟨 2. DAUX (Defines AOSP only, Uses vendor APIs)

Scenario: You don't add any new functions, but you internally call a vendor codec for speed.

```c
// AOSP API remains same
int decode_jpeg(const uint8_t* buf, int size) {
    // call into vendor optimized decoder
    return vendor_jpeg_decode(buf, size);
}
```

`vendor_jpeg_decode` is defined in `libvendorcodec.so`.

🔧 **Result**:

- Defines only AOSP functions → **DA**
- Uses vendor-only internal APIs → **UX**

📁 **Installed in**:

```bash
/vendor/lib[64]/vndk/libjpeg.so
```

🟦 3. DXUA (Defines new APIs, Uses only AOSP)

Scenario: You extend `libexif.so` by adding a helper function that AOSP didn't provide.

```c
// AOSP still works
int get_exif_data(...);

// Your extension
int get_camera_make_from_exif(const char* path) {
    // use AOSP functions internally
}
```

No vendor dependencies.

🔧 **Result**:

- Defines new function → **DX**
- Uses only AOSP APIs → **UA**

📁 **Installed in**:

```bash
/vendor/lib[64]/vndk/libexif.so
```

🟥 4. DXUX (Defines new APIs, Uses vendor APIs)

Scenario: You add a new API in `libjpeg.so` that calls another vendor helper.

```c
// Extended API
int get_vendor_debug_jpeg_info(const char* path) {
    return get_vendor_metadata(path); // From libvendorinfo.so
}
```

This adds:

- A new function → **DX**
- Uses a vendor-only lib → **UX**

🔧 **Result**: Full extension.

📁 **Installed in**:

```bash
/vendor/lib[64]/vndk/libjpeg.so
```

📛 What NOT to do

❌ Modify LL-NDK library like `liblog.so`

LL-NDK libraries must **never be extended** or altered.

If you do this:

```c
int __android_log_write(...) {
    // change behavior or hook it
```

```
}
```

❌ This is **NOT allowed**, because both system and vendor **must share** this library.

✅ How to Use Extensions Correctly

Let's say you created a vendor extension called `libjpeg_ext.so`, which adds a new function `get_vendor_debug_jpeg_info()`.

You do this in `Android.bp`:

```bp
cc_library {
    name: "libjpeg",  // AOSP version
    vendor_available: true,
    vndk: {
        enabled: true,
    },
}

cc_library {
    name: "libjpeg_ext", // Your vendor version
    vendor: true,
    vndk: {
        enabled: true,
        extends: "libjpeg",
    },
    shared_libs: [
        "libvendorinfo",
    ],
}
```

📁 This installs your custom `libjpeg.so` into:

```bash
/vendor/lib64/vndk/libjpeg.so
```

Any vendor binary that wants to use it:

```bp
cc_binary {
    name: "jpeg_tool",
    vendor: true,
    shared_libs: ["libjpeg_ext"],
}
```

🧠 Summary Table (with examples)

Type	What it means	Example	Uses vendor code?	Adds new APIs?	Installed in
DAUA	Same API, same deps	Optimized `libcrypto.so`	No	No	`/vendor/lib64/vndk/libcrypto.so`
DAUX	Same API, vendor deps	`libjpeg.so` calls `libvendorcodec.so`	Yes	No	`/vendor/lib64/vndk/libjpeg.so`
DXUA	New API, AOSP deps only	`libexif.so` with helper `get_camera_make_from_exif()`	No	Yes	`/vendor/lib64/vndk/libexif.so`
DXUX	New API and vendor deps	`libjpeg.so` with `get_vendor_debug_jpeg_info()`	Yes	Yes	`/vendor/lib64/vndk/libjpeg.so`

Would you like a **diagram** for this or a sample directory tree to show where everything installs?

8. VNDK snapshot design

bookmark_border
VNDK snapshots can be used by a system image to provide the correct VNDK libraries to vendor images even when system and vendor images are built from different versions of Android. Creating a VNDK snapshot requires capturing VNDK libraries as a snapshot and marking them with a version number. The vendor image may link with a specific VNDK version that provides required ABIs for the modules in the vendor image. However, within the same VNDK version, the VNDK libraries must be ABI-stable.

VNDK snapshot design includes methods for generating the pre-builds of a VNDK snapshot from the current system image and installing those pre-built libs to the system partition of a newer Android version.

About VNDK libraries
HIDL-HALs, introduced in Android 8.0, enables separate upgrades for system and vendor partitions. VNDK defines sets of libraries (VNDK-core, VNDK-SP and LL-NDK) that vendor code can link with and blocks the vendors from using libraries that are not in a VNDK set. As a result, the vendor image can be built and run if the proper VNDK sets on the system image are provided to the vendor image.

Note: For details on these libraries, refer to VNDK concepts.
VNDK-core
The VNDK-core set of libraries is installed in /system/lib[64]/vndk-${VER} and is available only for vendor processes with the API level equal to ${VER}. System processes may not use these libraries and must instead use the libraries installed in /system/lib[64]. Because of the strict namespace restriction for each process, the VNDK-core libraries are safe from dual-loading.

To include a library in VNDK-core, add the following to Android.bp:

```
vendor_available: true,
vndk: {
    enabled: true,
},
```

Note: If a system process loads library foo.so from system/lib and loads another foo.so from system/lib/vndk, foo.so is dual-loaded. Normally it is unsafe to load the same library twice in a process.
VNDK-SP
VNDK-SP libraries are installed in /system/lib[64]/vndk-sp-${VER} and are available to vendor processes and system processes (through the SP-HAL libraries installed in vendor partition). VNDK-SP libraries may be dual-loaded.

To include a library in VNDK-SP, add the following to Android.bp:

```
vendor_available: true,
vndk: {
    enabled: true,
    support_system_process: true,
},
```

LL-NDK
LL-NDK libraries are installed in /system/lib[64]. Vendor modules can use LL-NDK stub libraries to access pre-selected symbols of LL-NDK libraries. LL-NDK libraries must be backward-compatible and ABI-stable to enable old versions of vendor modules to use new versions of LL-NDK libraries. Because of the ABI-stable characteristics of LL-NDK, the VNDK snapshot does not need to include LL-NDK libraries for old vendor images.

About VNDK snapshots
Android 8.1 included VNDK libraries built from the source code. However, for later versions of Android, each VNDK version must be captured as a snapshot and provided as a pre-build to enabling linking to an older vendor image.

Starting in Android 9, new versions of Android will include at least one snapshot of VNDK-core and VNDK-SP directories for older versions in the Android source code. At build time, required snapshots will be installed to /system/lib[64]/vndk-${VER} and /system/lib[64]/vndk-sp-${VER} (directories that can be used by the vendor partition), where ${VER} is the string variable that represents the version name of the VNDK snapshot.

As the VNDK snapshot libraries may differ for each VNDK version, the VNDK snapshot also includes the linker namespace configurations, installed as etc/ld.config.${VER}.txt, /etc/llndk.libraries.${VER}.txt, and /etc/vndksp.libraries.${VER}.txt.

Example: Upgrade system and vendor images
No snapshot required; build without additional configurations for VNDK snapshots.

Example: Upgrade system image only

Must include the VNDK snapshot and linker namespace configuration files for the vendor image in the system image. The linker namespace configuration files are automatically configured to search for VNDK libraries in /system/lib[64]/vndk-${VER} and /system/lib[64]/vndk-sp-${VER}.

Figure 1. Upgrading system only
Example: Upgrade system image, minor vendor image change
Building a vendor image against a VNDK snapshot is not yet supported, so you must build the vendor image separately with its original source code, then upgrade the system image as described in the previous example.

VNDK snapshot architecture
To make an Android 9 system image compatible with an Android 8.1 vendor image, the VNDK snapshot that matches the Android 8.1 vendor image must be provided with the Android 9 system image, as shown below:

Figure 2. VNDK snapshot architecture
The VNDK snapshot design includes the following methods:

Generating a snapshot for VNDK-core and VNDK-SP libraries. Android 9 includes a script you can use to make a snapshot of the current VNDK build. This script bundles all libraries in /system/lib[64]/vndk-28 and /system/lib[64]/vndk-sp-28 that were built with the current source as a VNDK snapshot, where 28 is the VNDK version of Android 9. The snapshot also includes the linker namespace configuration files /etc/ld.config.28.txt, /etc/llndk.libraries.28.txt, and /etc/vndksp.libraries.28.txt. The generated snapshot will be used with newer Android versions (higher than Android 9).
Installing pre-built VNDK-core and VNDK-SP libraries from a snapshot. In Android 9, a VNDK snapshot has a set of pre-built VNDK-core libraries and a set of VNDK-SP libraries, as well as linker namespace configuration files. When you provide a list of VNDK snapshot versions to be installed, at build time, the system image installs the VNDK snapshot libraries to /system/lib[64]/vndk-${VER} and the /system/lib[64]/vndk-sp-${VER} directories and linker namespace configuration files for those VNDK snapshots to /etc directory.
VNDK versioning
Each Android release has only one VNDK snapshot and the SDK version is used as a VNDK version (which means the VNDK version has an integer number, such as 27 for Android 8.1). The VNDK version is fixed when the Android version is released. The VNDK version used by the vendor partition is stored automatically in the ro.vndk.version property, which can be read on runtime. This version is then used in identifying the vendor VNDK version for some libraries and identifying the VNDK snapshot version for namespace configuration.

Build VNDK libraries
The make vndk command builds libraries that have vndk: { enabled: true, … }, including dependencies and namespace configuration files. If BOARD_VNDK_VERSION := current is set, these libraries are built with the make command.

Because this build does not install the VNDK libraries from the snapshot, the installed VNDK libraries are not ABI-stable. However, when an Android version is released, the ABI for the current VNDK version is fixed. At this point, any ABI breakage is a build error, so patches to the Android version must not change the ABI for VNDK libraries.

Here's a **clear explanation of VNDK snapshot design**, with **real-world examples**, diagrams (described), and simple analogies.

🔧 WHY VNDK SNAPSHOTS?

Android wants to:

- Allow **system and vendor images to be upgraded separately**.
- Ensure that when system upgrades (e.g. Android 9 → 10), it can **still run the vendor code** (e.g. vendor built on Android 9) **without rebuild**.

💡 **But there's a challenge:**

> Vendor binaries depend on certain system libraries. If those system libraries change too much (ABI changes), the vendor binaries may crash.

📦 **Solution: VNDK Snapshots**

🔁 BASIC IDEA

> Freeze & capture a set of **VNDK libraries** at a given Android version (say Android 9 = VNDK v28) and include them in **newer system images** (e.g. Android 10).

This makes the newer Android system compatible with older vendor binaries.

✅ KEY VNDK LIBRARY TYPES

Library Type	Installed At	Used By	Special Notes
VNDK-core	`/system/lib[64]/vndk-28/`	Vendor only	Must be ABI stable, no dual-load
VNDK-SP	`/system/lib[64]/vndk-sp-28/`	System + Vendor	May be dual-loaded safely
LL-NDK	`/system/lib[64]/` (normal)	System + Vendor	Stable stub APIs, always compatible

🧠 ANALOGY

Think of VNDK snapshot as a **frozen compatibility layer**.

Imagine:

- Your vendor created a **binary driver** for Android 9.
- Android 10 wants to use it.
- It cannot recompile the driver (vendor binary).
- So Android 10 brings with it a **"compatibility box"**: VNDK snapshot from Android 9.

🗂️ VNDK SNAPSHOT CONTENTS

A VNDK snapshot includes:

- Prebuilt `.so` libraries (`libexample.so`, `libcamera.so`, etc.)
- Namespace config files:
  - `/etc/ld.config.28.txt`
  - `/etc/llndk.libraries.28.txt`
  - `/etc/vndksp.libraries.28.txt`

🛠️ These files tell the **dynamic linker** where to find the correct version of libraries at runtime.

📈 EXAMPLE USE CASES

✅ Case 1: System and Vendor both updated

No need for snapshot.

Everything built from source with matching versions.

✅ Case 2: System updated, Vendor image remains

You upgraded from Android 9 → 10, but want to reuse the old vendor image (built for Android 9).

✅ You include:

```bash
/system/lib64/vndk-28/libfoo.so
/system/lib64/vndk-sp-28/libbar.so
/etc/ld.config.28.txt
```

Now, when the vendor binary requests `libfoo.so`, the system gives it the VNDK 28 version.

🎯 **Your vendor image works without change**.

🔃 HOW SNAPSHOTS ARE CREATED

You can create a VNDK snapshot by:

```bash
make vndk
```

It captures:

- All current VNDK-core and VNDK-SP libraries
- Their versioned directories (e.g. **`vndk-28`**)
- All required **`.txt`** namespace files

📦 This snapshot can now be used in **future Android builds** to support old vendor images.

🔢 VNDK VERSIONING

Android Version	VNDK Version
Android 8.1	27
Android 9.0	28
Android 10	29
Android 11	30

Each snapshot is tagged with the version (e.g., **`vndk-28`**), and the vendor image includes:

```bash
ro.vndk.version=28
```

The dynamic linker uses this to load the correct libraries.

🧩 VNDK Snapshot Diagram (Described)

```pgsql
          +----------------------+
          | Android 10 System    |
          | ------------------   |
          | /system/lib64/       |
          | /vndk-28/            | <--- VNDK-core snapshot from Android 9
          | /vndk-sp-28/         | <--- VNDK-SP snapshot from Android 9
          | /etc/ld.config.28.txt|
          +----------------------+

                  || uses ||
                  \/

        +-------------------------+
        | Android 9 Vendor Image  |
        | Uses ro.vndk.version=28 |
        | Calls VNDK-28 libs      |
        +-------------------------+
```

🛑 KEY RULES

- **Snapshots are ABI-frozen**: Once Android 10 ships with VNDK-28, you **cannot change** the ABI of VNDK-28 libs anymore.
- LL-NDK libs are **always compatible** and don't need snapshots.
- Snapshots ensure **cross-version compatibility**.

🧪 Testing

- **VNDK ABI Checker** validates that the snapshot **doesn't change ABI** between releases.
- Any incompatible change = ❌ Build error.

✅ Summary

Feature	Description
What is VNDK Snapshot?	Prebuilt, versioned libraries from older Android used by newer system images
Why use it?	To support **older vendor binaries** after system upgrade
What’s inside?	`lib*.so` + linker configs like `ld.config.*.txt`
Where installed?	`/system/lib[64]/vndk-28/`, `/etc/ld.config.28.txt`
How to generate?	`make vndk`
Versioning?	Snapshot is tagged by Android SDK version (e.g., `vndk-28`)

Would you like a real **directory layout**, or how to **build with a snapshot manually**?

9. Generate vendor snapshots

bookmark_border
Android 11 supports VNDK snapshot build artifacts and vendor snapshots, which you can use to build vendor.img regardless of the Android version on the source tree. This enables mixed versions of images, such as an older vendor and a newer system image.

Mixed image versions aren't supported for the following.

Android.mk. Because Soong generates the vendor snapshot, modules defined in Android.mk aren't captured as a vendor snapshot (SoC-proprietary modules in Android.mk also aren't guaranteed to work).

Sanitizer. Vendor and VNDK snapshots don't support sanitizer as sanitizer variants need to be built separately.

About vendor snapshots
A vendor snapshot is an OEM-owned snapshot. It's a set of prebuilt C++ modules installed in /vendor but maintained on AOSP. Without capturing a vendor snapshot of the previous Android version, upgrading to a new Android version might break the vendor partition because vendor snapshot modules can be removed or changed without API or ABI compatibility.

A vendor snapshot contains following modules on AOSP.

Shared, static, and header libraries with vendor: true or vendor_available: true
VNDK static libraries with vendor_available: true
Executables and object files with vendor: true or vendor_available: true
Modules under the following directories are considered SoC-owned hardware code and are ignored.

device/
vendor/
hardware/, except for
hardware/interfaces
hardware/libhardware
hardware/libhardware_legacy
SoC-owned hardware codes might also exist in other directories. At this time, vendor snapshots don't support such configurations.

Build vendor snapshots
Build a vendor snapshot using the following commands.

```
. build/envsetup.sh
lunch target
m dist vendor-snapshot
```

These commands create a vendor-$(TARGET_DEVICE).zip file in $DIST_DIR. The following example shows a vendor snapshot zip file:

```
vendor-$(TARGET_DEVICE).zip
├── arch-arm64-armv8-a
│   ├── binary       -> binary files, *.json files
│   ├── header       -> *.json files
│   ├── object       -> *.o files, *.json files
│   ├── shared       -> *.so files, *.json files
│   └── static       -> *.a files, *.json files
├── arch-arm-armv8-a  -> (arch-arm64-armv8-a)
├── configs          -> *.rc files, *.xml files
├── include          -> exported header files (*.h, *.hh, etc.)
└── NOTICE_FILES     -> license txt files
```

The JSON files contain flags such as module name, exported directories, init_rc files, and vintf_fragments files.
The configs directory contains .rc and .xml files from flags init_rc and vintf_fragments.
Install vendor snapshots
Note: For details on uploading VNDK snapshots, see Using locally built VNDK snapshots.
To install a vendor snapshot, navigate to the destination directory and use the following commands.

```
python3 development/vendor_snapshot/update.py --local $DIST_DIR --install-dir \
    vendor/vendor name/vendor_snapshot VER
```

This command generates /vendor/vendor name/vendor_snapshot/vVER/$(TARGET_ARCH)/Android.bp. The following example shows the directory structure of a vendor snapshot:

```
vendor/vendor name/vendor_snapshot/
├── v30
│   ├── arm64
│   │   ├── arch-arm64-armv8-a
│   │   │   ├── binary         -> (prebuilt libs)
│   │   │   ├── object         -> (prebuilt libs)
│   │   │   ├── shared         -> (prebuilt libs)
│   │   │   └── static         -> (prebuilt libs)
│   │   ├── arch-arm-armv8-a   -> (same as above)
│   │   ├── configs            -> (config files)
│   │   ├── include            -> (exported header files)
│   │   ├── NOTICE_FILES       -> (license files)
│   │   └── Android.bp         -> (snapshot modules)
│   ├── arm                    -> (same as above)
│   ├── x86_64                 -> (same as above)
│   └── x86                    -> (same as above)
└── (other vendor versions)    -> (same as above)
```

Build vendor images
Using VNDK snapshot build artifacts and a vendor snapshot, you can build vendor.img regardless of the Android version on the source tree, making it possible to create mixed versions of images. After installing both the VNDK snapshot and vendor snapshot, set BOARD_VNDK_VERSION to the vendor's version. For example:

```
# BoardConfig.mk
BOARD_VNDK_VERSION := 29
```

When BOARD_VNDK_VERSION is set to a value other than current, the build system:

Enables VNDK snapshot modules and vendor snapshot modules of BOARD_VNDK_VERSION. Each snapshot module overrides a C++ source module having the same name.
Sets ro.vndk.version to BOARD_VNDK_VERSION.
Installs VNDK snapshot modules of BOARD_VNDK_VERSION to system_ext.

Here's a **simple explanation of Vendor Snapshots in Android**, with **real-world examples and use cases**, so you can understand how they work and when to use them — especially when building vendor images independently of the current AOSP version.

🔧 What Is a Vendor Snapshot?

A **Vendor Snapshot** is:

- A **prebuilt set of C++ libraries and binaries** from `/vendor`.
- Generated from an older Android version.
- Used to **build vendor.img** even if the **system image is from a newer Android version**.

💡 It makes **cross-version compatibility** possible.

🤔 Why Do We Need It?

Imagine this scenario:

You are a **SoC vendor** (e.g., Qualcomm), and you:

- Built your vendor binaries on Android 10 (VNDK v29).
- Now a phone company wants to use **Android 11 (VNDK v30)** system image with your **old vendor binaries**.

This can break compatibility because:

> APIs in the vendor binaries might not match the new system.

🛠️ **Solution:**

- Create a **VNDK snapshot** from Android 10 (v29)
- Create a **Vendor snapshot** from your old vendor binaries (v29)
- Use both to build vendor.img again — even while the system is now on Android 11.

🔗 Relation Between Snapshots

Snapshot Type	Used For	Source From	Used In
VNDK Snapshot	System libraries for vendor	Built from AOSP system	`/system`
Vendor Snapshot	Vendor binaries for reuse	Built from vendor code	`/vendor`

📦 What's Inside a Vendor Snapshot?

Example snapshot zip file:

```
pgsql

vendor-mydevice.zip
├── arch-arm64-armv8-a/
│   ├── binary/         <- compiled executables (.bin, .json)
│   ├── header/         <- .json metadata
│   ├── object/         <- object files (.o, .json)
│   ├── shared/         <- .so prebuilt shared libraries
│   ├── static/         <- .a static libraries
├── configs/            <- .rc, .xml init files
├── include/            <- exported header files (.h)
└── NOTICE_FILES/       <- license texts
```

🏗️ How to Generate Vendor Snapshot

Step 1: Build the snapshot

```
bash

. build/envsetup.sh
lunch <your_target>
m dist vendor-snapshot
```

Output:

```
bash

$DIST_DIR/vendor-mydevice.zip
```

📥 How to Install Vendor Snapshot

Step 2: Extract & Install

```
bash

python3 development/vendor_snapshot/update.py \
  --local $DIST_DIR \
  --install-dir vendor/myvendor/vendor_snapshot/29
```

This creates:

```
vbnet

vendor/myvendor/vendor_snapshot/
└── v29/
    └── arm64/
        ├── arch-arm64-armv8-a/
        │   ├── shared/
        │   ├── static/
        │   ├── object/
        ├── configs/
```

```
        ├── include/
        └── Android.bp  <- Soong file to register prebuilt modules
```

This layout mimics the actual vendor partition.

🛠️ How to Build `vendor.img` Using Snapshot

Step 3: Configure the board to use snapshot version

In your **`BoardConfig.mk`**:

```make
BOARD_VNDK_VERSION := 29
```

This tells Soong:

- Use **VNDK snapshot v29**
- Use **Vendor snapshot v29**
- Set **`ro.vndk.version=29`**

⛓️ So all linking and partition-building use the same "frozen" versions.

🧪 Result: Mixed Version Build

Partition	Source Android Version	Snapshot Used	Notes
`/system`	Android 11 (v30)	VNDK v29	Supports backward compatibility
`/vendor`	Android 10 (v29)	Vendor v29	Uses prebuilt vendor snapshot

✅ Builds **clean vendor.img** on Android 11, even though it uses Android 10 vendor code.

🧱 Which Modules Are Included in Vendor Snapshot?

✅ **Included** (if **`vendor: true`** or **`vendor_available: true`**)

- **`.so`** shared libs
- **`.a`** static libs
- **`.h`** headers
- **`.rc`**, **`.xml`** config/init files
- Executables and **`.o`** object files

❌ **Excluded**

- Anything defined in **`Android.mk`**
- SoC-proprietary modules inside **`/device`**, **`/vendor`**, **`/hardware`** (except **`hardware/interfaces`** and libhardware)

🛑 Limitations

Limitation	Reason
❌ No **`Android.mk`**	Only Soong (**`Android.bp`**) modules are snapshotted
❌ No Sanitizers	e.g., **`asan`**, **`ubsan`** not supported inside snapshot builds
❌ Not all SoC code is included	Some directories are excluded by default

✅ Summary

Feature	Details
What is it?	Prebuilt set of vendor libraries used across Android versions

Feature	Details
Why use it?	To build **`vendor.img`** even when source tree is a newer Android version
How to build?	**`m dist vendor-snapshot`**
How to use?	**`python3 update.py …`**, then set **`BOARD_VNDK_VERSION := <ver>`**
What's inside?	Shared libs, headers, objects, static libs, configs
Not supported?	**`Android.mk`**, sanitizers, some proprietary SoC code

---

Would you like a step-by-step example with code for generating and installing a vendor snapshot for a device like **`pixel_5`**?

10. Linker namespace

bookmark_border
The dynamic linker tackles two challenges in Treble VNDK design:

SP-HAL shared libraries and their dependencies, including VNDK-SP libraries, are loaded into framework processes. There should be some mechanisms to prevent symbol conflicts.
dlopen() and android_dlopen_ext() can introduce some runtime dependencies that aren't visible at build time and can be difficult to detect using static analysis.
These two challenges can be resolved by the linker namespace mechanism. This mechanism is provided by the dynamic linker. It can isolate the shared libraries in different linker namespaces so that libraries with same library name but with different symbols won't conflict.

On the other hand, the linker namespace mechanism provides the flexibility so that some shared libraries can be exported by a linker namespace and used by another linker namespace. These exported shared libraries can become application programming interfaces that are public to other programs while hiding their implementation details within their linker namespaces.

For example, /system/lib[64]/libcutils.so and /system/lib[64]/vndk-sp-${VER}/libcutils.so are two shared libraries. These two libraries can have different symbols. They're loaded into different linker namespaces so that framework modules can depend on /system/lib[64]/libcutils.so and SP-HAL shared libraries can depend on /system/lib[64]/vndk-sp-${VER}/libcutils.so.

On the other hand, /system/lib[64]/libc.so is an example of a public library that's exported by a linker namespace and imported into many linker namespaces. The dependencies of /system/lib[64]/libc.so, such as libnetd_client.so, are loaded into the namespace in which /system/lib[64]/libc.so resides. Other namespaces won't have accesses to those dependencies. This mechanism encapsulates the implementation details while providing the public interfaces.

How it works
The dynamic linker is responsible for loading the shared libraries specified in DT_NEEDED entries or the shared libraries specified by the argument of dlopen() or android_dlopen_ext(). In both cases, the dynamic linker finds the linker namespace where the caller resides and tries to load the dependencies into the same linker namespace. If the dynamic linker can't load the shared library into the specified linker namespace, it asks the linked linker namespace for exported shared libraries.

Configuration file format
The configuration file format is based on the INI file format. A typical configuration file looks like this:

```
dir.system = /system/bin
dir.system = /system/xbin
dir.vendor = /vendor/bin

[system]
additional.namespaces = sphal,vndk

namespace.default.isolated = true
namespace.default.search.paths = /system/${LIB}
namespace.default.permitted.paths = /system/${LIB}/hw
namespace.default.asan.search.paths = /data/asan/system/${LIB}:/system/${LIB}
namespace.default.asan.permitted.paths = /data/asan/system/${LIB}/hw:/system/${LIB}/hw

namespace.sphal.isolated = true
namespace.sphal.visible = true
namespace.sphal.search.paths = /odm/${LIB}:/vendor/${LIB}
namespace.sphal.permitted.paths = /odm/${LIB}:/vendor/${LIB}
namespace.sphal.asan.search.paths  = /data/asan/odm/${LIB}:/odm/${LIB}
namespace.sphal.asan.search.paths += /data/asan/vendor/${LIB}:/vendor/${LIB}
namespace.sphal.asan.permitted.paths  = /data/asan/odm/${LIB}:/odm/${LIB}
namespace.sphal.asan.permitted.paths += /data/asan/vendor/${LIB}:/vendor/${LIB}
namespace.sphal.links = default,vndk
namespace.sphal.link.default.shared_libs = libc.so:libm.so
namespace.sphal.link.vndk.shared_libs = libbase.so:libcutils.so

namespace.vndk.isolated = true
namespace.vndk.search.paths = /system/${LIB}/vndk-sp-29
namespace.vndk.permitted.paths = /system/${LIB}/vndk-sp-29
```

namespace.vndk.links = default
namespace.vndk.link.default.shared_libs = libc.so:libm.so

[vendor]
namespace.default.isolated = false
namespace.default.search.paths = /vendor/${LIB}:/system/${LIB}
The configuration file includes:

Several directory-section mapping properties at the beginning for the dynamic linker to select the effective section.
Several linker namespaces configuration sections:
Each section contains several namespaces (graph vertexes) and several fallback links between namespaces (graph arcs).
Each namespace has its own isolation, search paths, permitted paths, and visibility settings.
The tables below describe the meaning of each property in detail.

Directory-section mapping property
Property Description Example
dir.name

A path to a directory that the [name] section applies to.

Each property maps the executables under the directory to a linker namespaces configuration section. There might be two (or more) properties that have the same name but point to different directories.

dir.system = /system/bin
dir.system = /system/xbin
dir.vendor = /vendor/bin

This indicates that the configuration specified in the [system] section applies to the executables that are loaded from either /system/bin or /system/xbin.

The configuration specified in the [vendor] section applies to the executables that are loaded from /vendor/bin.

Relation properties
Property Description Example
additional.namespaces
A comma-separated list of additional namespaces (in addition to the default namespace) for the section.

additional.namespaces = sphal,vndk

This indicates that there are three namespaces (default, sphal, and vndk) in the [system] configuration.

namespace.name.links
A comma-separated list of fallback namespaces.

If a shared library can't be found in the current namespace, the dynamic linker tries to load the shared library from the fallback namespaces. The namespace specified at the beginning of the list has higher priority.

Note: Fallback namespaces aren't transitive. For example, namespace A links to namespace B and namespace B links to namespace C. If the dynamic linker can't find the library in namespace A, it searches ONLY namespace B. It doesn't search namespace C.
namespace.sphal.links = default,vndk

If a shared library or an executable requests a shared library that can't be loaded into the sphal namespace, the dynamic linker tries to load the shared library from the default namespace.

And then, if the shared library can't be loaded from the default namespace either, the dynamic linker tries to load the shared library from the vndk namespace.

Finally, if all attempts fail, the dynamic linker returns an error.

namespace.name.link.other.shared_libs
A colon-separated list of shared libraries that can be searched in the other namespaces when those libraries can't be found in the name namespace.

This property can't be used with namespace.name.link.other.allow_all_shared_libs.

Note: This property shares the same underlying implementation with public.libraries.txt files. Both mechanisms control the imported shared libraries by specifying a link with a library name filter.

The difference is that ld.config.txt is loaded by the dynamic linker and all namespaces are created when a program starts. In contrast, libnativeloader creates a linker namespace when the Zygote process is forked and specialized for an app. The namespace for the app native libraries has a link that allows only the library names specified in public.libraries.txt.

namespace.sphal.link.default.shared_libs = libc.so:libm.so

This indicates that the fallback link accepts only libc.so or libm.so as the requested library name. The dynamic linker ignores the fallback link from sphal to default namespace if the requested library name isn't libc.so or libm.so.

namespace.name.link.other.allow_all_shared_libs
A boolean value that indicates whether all shared libraries can be searched in the other namespace when those libraries can't be found in the name namespace.

This property can't be used with namespace.name.link.other.shared_libs.

namespace.vndk.link.sphal.allow_all_shared_libs = true

This indicates that all library names can walk through the fallback link from vndk to sphal namespace.

Namespace properties
Property Description Example
namespace.name.isolated
A boolean value that indicates whether the dynamic linker should check where the shared library resides.

If isolated is true, only the shared libraries that are in one of the search.paths directories (excluding subdirectories) or are under one of the permitted.paths directories (including subdirectories) can be loaded.

If isolated is false (default), the dynamic linker doesn't check the path of shared libraries.

namespace.sphal.isolated = true

This indicates that only the shared libraries in search.paths or under permitted.paths can be loaded into the sphal namespace.

namespace.name.search.paths
A colon-separated list of directories to search for shared libraries.

The directories specified in search.paths are prepended to the requested library name if function calls to dlopen() or DT_NEEDED entries don't specify the full path. The directory specified at the beginning of the list has higher priority.

When isolated is true, shared libraries that are in one of the search.paths directories (excluding subdirectories) can be loaded regardless of the permitted.paths property.

For example, if search.paths is /system/${LIB} and permitted.paths is empty, /system/${LIB}/libc.so can be loaded but /system/${LIB}/vndk/libutils.so can't be loaded.

Note: ${LIB} is a built-in placeholder. It's replaced by lib for 32-bit processes and lib64 for 64-bit processes.
namespace.default.search.paths = /system/${LIB}

This indicates that the dynamic linker searches /system/${LIB} for shared libraries.

namespace.name.asan.search.paths
A colon-separated list of directories to search for shared libraries when AddressSanitizer (ASan) is enabled.

namespace.name.search.paths is ignored when ASan is enabled.

namespace.default.asan.search.paths = /data/asan/system/${LIB}:/system/${LIB}

This indicates that when ASan is enabled the dynamic linker searches /data/asan/system/${LIB} first and then searches /system/${LIB}.

namespace.name.permitted.paths
A colon-separated list of directories (including subdirectories) where the dynamic linker can load the shared libraries (in addition to search.paths) when isolated is true.

The shared libraries that are under the subdirectories of permitted.paths can also be loaded. For example, if permitted.paths is /system/${LIB}, both /system/${LIB}/libc.so and /system/${LIB}/vndk/libutils.so can be loaded.

If isolated is false, permitted.paths are ignored and a warning is emitted.

Note: permitted.paths are NOT prepended to the requested library names when the dynamic linker is searching for a shared library. Its main purpose is to allow programers to load shared libraries into an isolated namespace by specifying the full path of the shared library.
namespace.default.permitted.paths = /system/${LIB}/hw

This indicates that the shared libraries under /system/${LIB}/hw can be loaded into the isolated default namespace.

For example, without permitted.paths, libaudiohal.so can't load /system/${LIB}/hw/audio.a2dp.default.so into the default namespace.

namespace.name.asan.permitted.paths
A colon-separated list of directories where the dynamic linker can load the shared libraries when ASan is enabled.

namespace.name.permitted.paths is ignored when ASan is enabled.

namespace.default.asan.permitted.paths = /data/asan/system/${LIB}/hw:/system/${LIB}/hw

This indicates that when ASan is enabled shared libraries under /data/asan/system/${LIB}/hw or /system/${LIB}/hw can be loaded to the isolated default namespace.

namespace.name.visible
A boolean value that indicates whether the program (other than libc) can obtain a linker namespace handle with android_get_exported_namespace() and open a shared library in the linker namespace by passing the handle to android_dlopen_ext().

If visible is true, android_get_exported_namespace() always returns the handle if the namespace exists.

If visible is false (default), android_get_exported_namespace() always returns NULL regardless of the presence of the namespace. Shared libraries can be loaded into this namespace only if (1) they're requested by another linker namespace that has a fallback link to this namespace, or (2) they're requested by other shared libraries or executables in this namespace.

namespace.sphal.visible = true

This indicates that android_get_exported_namespace("sphal") can return a valid linker namespace handle.

Linker namespace creation
In Android 11, linker configuration is created at runtime under /linkerconfig instead of using plain text files in ${android-src}/system/core/rootdir/etc. The configuration is generated at boot time based on the runtime environment, which includes following items:

If device supports VNDK
Vendor partition's target VNDK version
Product partition's VNDK version
Installed APEX modules
Linker configuration is created by resolving dependencies between linker namespaces. For example, if there are any updates on the APEX modules that include dependency updates, linker configuration is generated reflecting these changes. More details to create linker configuration can be found in ${android-src}/system/linkerconfig.

Linker namespace isolation
There are three configuration types. Depending on the value of PRODUCT_TREBLE_LINKER_NAMESPACES and BOARD_VNDK_VERSION in BoardConfig.mk, the corresponding configuration is generated at boot time.

PRODUCT_TREBLE_
LINKER_NAMESPACES BOARD_VNDK_
VERSION Selected configuration VTS requirement
true current VNDK Mandatory for devices launched with Android 9 or higher
Empty VNDK Lite Mandatory for devices launched with Android 8.x
false Empty Legacy For non-Treble devices
VNDK Lite configuration isolates SP-HAL and VNDK-SP shared libraries. In Android 8.0, this must be the configuration file for dynamic linker when PRODUCT_TREBLE_LINKER_NAMESPACES is true.

VNDK configuration also isolates SP-HAL and VNDK-SP shared libraries. In addition, this configuration provides the full dynamic linker isolation. It ensures that modules in the system partition won't depend on the shared libraries in the vendor partitions and vice versa.

In Android 8.1 or higher, VNDK configuration is the default configuration and it's highly recommended to enable full dynamic linker isolation by setting BOARD_VNDK_VERSION to current.

VNDK configuration
Caution: This section includes library references to RenderScript. Starting with Android 12, the RenderScript APIs are deprecated. They will continue to function, but device and component manufacturers are expected to stop providing hardware-acceleration support over time. To take full advantage of GPU acceleration, AOSP recommends that you Migrate from RenderScript.
VNDK configuration isolates the shared library dependencies between the system partition and vendor partitions. Compared to configurations mentioned in the previous subsection, the differences are outlined as follows:

Framework processes

default, vndk, sphal, and rs namespaces are created.
All namespaces are isolated.
System shared libraries are loaded into the default namespace.
SP-HALs are loaded into the sphal namespace.
VNDK-SP shared libraries loaded into the vndk namespace.
Vendor processes

default, vndk, and system namespaces are created.
The default namespace is isolated.
Vendor shared libraries are loaded into the default namespace.
VNDK and VNDK-SP shared libraries are loaded into the vndk namespace.
LL-NDK and its dependencies are loaded into the system namespace.
The relationship between the linker namespaces is illustrated below.

Linker namespace graph described in VNDK configuration
Figure 1. Linker namespace isolation (VNDK configuration).

In the image above, LL-NDK and VNDK-SP stand for following shared libraries:

LL-NDK
libEGL.so
libGLESv1_CM.so
libGLESv2.so
libGLESv3.so
libandroid_net.so
libc.so
libdl.so
liblog.so
libm.so
libnativewindow.so
libneuralnetworks.so
libsync.so
libvndksupport.so
libvulkan.so
VNDK-SP
android.hardware.graphics.common@1.0.so
android.hardware.graphics.mapper@2.0.so
android.hardware.renderscript@1.0.so
android.hidl.memory@1.0.so
libRSCpuRef.so
libRSDriver.so
libRS_internal.so
libbase.so
libbcinfo.so
libc++.so
libcutils.so
libhardware.so
libhidlbase.so
libhidlmemory.so
libhidltransport.so
libhwbinder.so
libion.so
libutils.so
libz.so
You can find more details in /linkerconfig/ld.config.txt from the device.

VNDK Lite configuration
As of Android 8.0, the dynamic linker is configured to isolate the SP-HAL and VNDK-SP shared libraries such that their symbols don't conflict with other framework shared libraries. The relationship between the linker namespaces is shown below.

Linker namespace graph described in VNDK Lite configuration
Figure 2. Linker namespace isolation (VNDK Lite configuration)
LL-NDK and VNDK-SP stand for following shared libraries:

LL-NDK
libEGL.so
libGLESv1_CM.so
libGLESv2.so
libc.so
libdl.so
liblog.so
libm.so
libnativewindow.so
libstdc++.so (not in the configuration)
libsync.so
libvndksupport.so
libz.so (moved to VNDK-SP in the configuration)
VNDK-SP
android.hardware.graphics.common@1.0.so
android.hardware.graphics.mapper@2.0.so
android.hardware.renderscript@1.0.so
android.hidl.memory@1.0.so
libbase.so
libc++.so
libcutils.so
libhardware.so
libhidlbase.so
libhidlmemory.so
libhidltransport.so
libhwbinder.so
libion.so
libutils.so
The table below lists the namespaces configuration for framework processes, which is excerpted from the [system] section in the VNDK Lite configuration.

Namespace Property Value
default search.paths /system/${LIB}
/odm/${LIB}
/vendor/${LIB}
/product/${LIB}
isolated false
sphal search.paths /odm/${LIB}
/vendor/${LIB}
permitted.paths /odm/${LIB}
/vendor/${LIB}
isolated true
visible true
links default,vndk,rs
link.default.shared_libs LL-NDK
link.vndk.shared_libs VNDK-SP
link.rs.shared_libs libRS_internal.so
vndk (for VNDK-SP) search.paths /odm/${LIB}/vndk-sp
/vendor/${LIB}/vndk-sp
/system/${LIB}/vndk-sp-${VER}
permitted.paths /odm/${LIB}/hw
/odm/${LIB}/egl
/vendor/${LIB}/hw
/vendor/${LIB}/egl
/system/${LIB}/vndk-sp-${VER}/hw
isolated true
visible true
links default

link.default.shared_libs LL-NDK
rs (for RenderScript) search.paths /odm/${LIB}/vndk-sp
/vendor/${LIB}/vndk-sp
/system/${LIB}/vndk-sp-${VER}
/odm/${LIB}
/vendor/${LIB}
permitted.paths /odm/${LIB}
/vendor/${LIB}
/data (for compiled RS kernel)
isolated true
visible true
links default,vndk
link.default.shared_libs LL-NDK
libmediandk.so
libft2.so
link.vndk.shared_libs VNDK-SP
The table below presents the namespaces configuration for vendor processes, which is excerpted from the [vendor] section in the VNDK Lite configuration.

Namespace Property Value
default search.paths /odm/${LIB}
/odm/${LIB}/vndk
/odm/${LIB}/vndk-sp
/vendor/${LIB}
/vendor/${LIB}/vndk
/vendor/${LIB}/vndk-sp
/system/${LIB}/vndk-${VER}
/system/${LIB}/vndk-sp-${VER}
/system/${LIB} (deprecated)
/product/${LIB} (deprecated)
isolated false
More details can be found in /linkerconfig/ld.config.txt from the device.

Document history
Android 11 Changes
In Android 11, the static ld.config.*.txt files are removed from the codebase and LinkerConfig generates them in runtime instead.
Android 9 changes
In Android 9, the vndk linker namespace is added to vendor processes and VNDK shared libraries are isolated from the default linker namespace.
Replace PRODUCT_FULL_TREBLE with more specific PRODUCT_TREBLE_LINKER_NAMESPACES.
Android 9 changes the names of the following dynamic linker configuration files.
Android 8.x Android 9 Description
ld.config.txt.in ld.config.txt For devices with runtime linker namespace isolation
ld.config.txt ld.config.vndk_lite.txt For devices with VNDK-SP linker namespace isolation
ld.config.legacy.txt ld.config.legacy.txt For legacy devices running Android 7.x or lower
Remove android.hardware.graphics.allocator@2.0.so.
product and odm partitions are added.

Here's a simplified and complete explanation of **linker namespaces** and how they work in Android, especially in the context of **VNDK, vendor snapshots, and Treble**:

🔧 Why Linker Namespaces Are Needed (Problem It Solves)

Android Treble separates the **system** and **vendor** partitions. This helps vendors (OEMs) update hardware-related code without needing to recompile everything with each Android release. But there's a problem:

- **Both system and vendor binaries might use a library with the same name** (e.g., `libcutils.so`) but with **different implementations** (i.e., different symbols).
- This causes **symbol conflicts or crashes**, especially when both versions are loaded into the same process.

🔒 Solution: Linker Namespaces

A **linker namespace** is like a sandbox for shared libraries. It:

1. **Isolates** libraries from one another to prevent conflicts.
2. Allows **controlled sharing** via explicit exports and links.

📦 Real Example: `libcutils.so`

System uses:	`/system/lib64/libcutils.so`
Vendor uses:	`/system/lib64/vndk-sp-29/libcutils.so`

Even though both are named `libcutils.so`, they live in **different linker namespaces**:

- System processes use the **default namespace**.
- Vendor HALs (SP-HALs) use the **sphal** and **vndk** namespaces.

This isolation avoids conflicts between AOSP and vendor code.

🗂️ How It Works

When an app or process runs and needs a shared library (e.g., `libbase.so`):

- The **dynamic linker** loads it based on the namespace rules.
- If the library isn't found in the process's current namespace, it tries fallback **linked namespaces** (if defined), but only for specific libraries.

🧱 Namespace Configuration (in `ld.config.*.txt` or `/linkerconfig`) Structure

Each section in the config maps to:

- **Process locations** (like `/system/bin`, `/vendor/bin`)
- **Namespace definitions**, which include:
  - `search.paths`: where to look for `.so` files
  - `permitted.paths`: full paths allowed if `isolated = true`
  - `links`: fallbacks to other namespaces
  - `shared_libs`: which libraries are allowed to be pulled in

🧭 Example: SP-HAL Uses VNDK-SP

```plaintext
[sphal]
namespace.sphal.isolated = true
namespace.sphal.search.paths = /vendor/lib64
namespace.sphal.links = default,vndk
namespace.sphal.link.default.shared_libs = libc.so:libm.so
namespace.sphal.link.vndk.shared_libs = libcutils.so:libbase.so
```

This means:

- SP-HAL code will search for `.so` files in `/vendor/lib64`.
- If it can't find a library, it will fall back to the `default` or `vndk` namespaces.
- But only **specific libraries** (like `libcutils.so`) are allowed through the fallback.

⚙️ Namespace Use in Vendor Processes

Vendor-side processes have access to:

- `default`: for vendor binaries.
- `vndk`: for AOSP VNDK libraries.
- `system`: for **LL-NDK** libraries (like `libc.so`, `liblog.so`).

System processes don't load from the `vendor` partition directly to maintain **modularity and stability**.

🧠 Types of Namespaces in Android

Namespace	Purpose
`default`	For main system libraries
`sphal`	For SP-HALs (vendor HALs in system)
`vndk`	For VNDK-SP and VNDK-core libraries
`system`	For LL-NDK in vendor processes
`rs`	For RenderScript (now deprecated)

🛠️ Linker Namespace in Action: A Simple Story

- **Vendor's camera HAL** (`vendor/lib64/camera_hal.so`) uses `libutils.so` → found in `/system/lib64/vndk-sp-29/`
- The HAL is in `sphal` namespace.
- The linker checks `sphal`, finds `libutils.so` in `vndk-sp`, and loads it **without touching** system's `libutils.so`.

📄 Runtime Linker Config (`/linkerconfig`)

Since **Android 11**, instead of static `ld.config.txt`, Android now generates **runtime linker configs** under `/linkerconfig` during boot, based on:

- VNDK version (`ro.vndk.version`)
- APEX modules
- Product/Vendor properties

This allows **dynamic reconfiguration** of linker behavior on the fly.

🚦 Summary Table

Concept	Description
Namespace	Isolated environment for loading `.so` files
Linking	Fallback chain for missing libraries
Isolation	Prevents symbol conflicts between system/vendor
Visibility	Allows `android_dlopen_ext()` to access namespaces
VNDK	Set of libraries available to vendor, split into SP, core, LL-NDK
/linkerconfig/	Dynamic runtime config files for linker namespaces

---

Would you like visual diagrams or a walk-through of how this plays out in an actual boot process?

11. explain in simple wods with examples

Sure! Let's break down **linker namespaces** in **very simple words** with **real-life-style examples** so you can easily understand them.

🌍 Imagine a Library Building with Many Rooms

Think of your Android device as a **big library building** full of **rooms**. Each **room** is like a **process** (like a system app, vendor service, or HAL). Inside each room, people (code) can **read books** (shared libraries like `.so` files).

Now, suppose:

- There are **many books with the same name** in different rooms (e.g., two `libcutils.so` books).
- Each room should only use **its own version** of the book to avoid confusion.

That's where **namespaces** come in — they are like **private book collections for each room**.

🚫 Problem Without Namespaces

Without namespaces:

- Two rooms use `libcutils.so`.
- One is from `/system/lib64/libcutils.so`
- Other is from `/vendor/lib64/libcutils.so`
- They might have **same name** but **different contents**!
- If both are loaded into one room, it can crash or behave incorrectly (like reading the wrong recipe!).

✅ How Linker Namespaces Help

Linker namespaces are like:

- "Only read from *your own shelf*, unless I say you can borrow from another shelf."

Each process (room) has:

- Its own set of `.so` files (books).
- **Rules**: What can it read, what can it borrow, and from where.

🧩 Real Example: System and Vendor Libraries

System App (`/system/bin/app`)

- Lives in `default` namespace.
- Loads libraries like `/system/lib64/libbase.so`.

Vendor HAL (`/vendor/bin/camera_hal`)

- Lives in `sphal` or `vndk` namespace.
- Loads `/system/lib64/vndk-sp-30/libutils.so`, not the system one!

👉 This keeps system and vendor code **separate and safe**.

🔄 Example of Borrowing (Linking)

The **camera HAL** needs `libbase.so`, but it's not in its shelf.

So we allow:

```ini
namespace.sphal.link.default.shared_libs = libbase.so
```

This means:

- "Hey sphal, if you need **`libbase.so`**, you're allowed to borrow it from **`default`**."

🔐 Visibility & Isolation

- **`isolated = true`**: Can't read outside your allowed shelves.
- **`visible = true`**: Other rooms can see your shelves.

Example:

```ini
namespace.sphal.isolated = true
namespace.sphal.search.paths = /vendor/lib64
namespace.sphal.permitted.paths = /vendor/lib64:/vendor/lib64/hw
namespace.sphal.visible = true
```

This means:

- Only read from **`/vendor/lib64`**.
- Others can "see" this shelf.

🏠 Home Example

Imagine:

- Mom has **her own bookshelf** (**`/system/lib64`**) — for cooking books.
- Dad has **his own bookshelf** (**`/vendor/lib64`**) — for engine repair books.

Even though both have a book named **"Tools"**:

- They mean **different things**.
- They **must not be swapped**.

So:

- Mom (system process) reads only her version.
- Dad (vendor process) reads his version.
- If Dad needs "Measuring Cups" (common), he's allowed to **borrow** from Mom's shelf **only that specific book**.

That's linker namespaces in Android.

📁 What Happens at Boot?

In Android 11+, a tool creates the final configuration in **`/linkerconfig/`** like:

- Which process uses which namespace
- What **`.so`** libraries they can access
- What they can borrow

This is done **dynamically**, based on installed APEX, VNDK version, etc.

💡 Summary in Simple Words

Term	Meaning (Simple)	Example
Linker Namespace	A private bookshelf for a process	**`/vendor/bin/camera_hal`** uses **`sphal`** namespace
Isolated = true	Only read from allowed shelves	**`sphal`** can't read **`/system/lib`** unless allowed
Link	A rule to borrow from another shelf	**`sphal`** can borrow **`libc.so`** from **`default`**
Visible = true	Let others see your bookshelf	Needed if shared between processes
VNDK	Set of approved **`.so`** books for vendor	Lives in **`/system/lib64/vndk-*`**
LL-NDK	Public system books vendors can always use	Like **`libc.so`**, **`liblog.so`**

Would you like me to draw a **diagram** or give a **real file example** from an Android device to make it even easier?

12. Directories, rules, and sepolicy

bookmark_border
This page describes the directory layout for devices running Android 8.0 and higher, VNDK rules, and associated sepolicy.

Directory layout
The Degenerated Directory Layout consists of the following directories:

/system/lib[64] contains all framework shared libraries, including LL-NDK, VNDK, and framework-only libraries (including LL-NDK-Private and some libraries with the same names as the ones in VNDK-SP).
/system/lib[64]/vndk-sp contains VNDK-SP libraries for same-process HALs.
/vendor/lib[64] contains the VNDK libraries extended by the vendor (either DXUA or DXUX VNDK libraries), same-process HAL implementations, and other vendor shared libraries.
/vendor/lib[64]/vndk-sp may contain the VNDK-SP libraries extended by the vendor.
Vendor modules load the VNDK libraries from /system/lib[64].

VNDK rules
This section provides a comprehensive list of VNDK rules:

Framework processes must not load non-SP-HAL shared libraries from vendor partitions (strictly enforced starting from Android 8.1).
Vendor processes must not load non-LL-NDK, non-VNDK-SP, and non-VNDK libraries from the system partition. (not strictly enforced in Android O but will be in a future release).
Note: To benefit from the framework-only OTA beyond Android 8.0, this rule must not be violated in devices launched with Android 8.0.
Installed VNDK libraries must be a subset of Google-defined eligible VNDK libraries.
The outer dependencies of SP-HAL and SP-HAL-Dep must be restricted to LL-NDK or Google-defined VNDK-SP libraries.
The dependencies of an SP-HAL shared library must be restricted to LL-NDK libraries, Google-defined VNDK-SP libraries, other SP-HAL libraries, and/or other vendor shared libraries that can be labeled as SP-HAL-Dep libraries.
A vendor shared library can be labeled as a SP-HAL-Dep library only if it is not an AOSP library and its dependencies are restricted to LL-NDK libraries, Google-defined VNDK-SP libraries, SP-HAL libraries, and/or other SP-HAL-Dep libraries.
VNDK-SP must be self-contained. libRS_internal.so gets special treatment in Android 8.0, but will be revisited in a future release.
No framework-vendor communication through non-HIDL interfaces, including (but not limited to) binder, sockets, shared memories, files, etc.
The size of the system partition must be large enough to contain two copies of all eligible VNDK libraries and a copy of ineligible framework shared libraries.
sepolicy
Framework processes described in this section correspond to coredomain in sepolicies while vendor processes correspond to non-coredomain. For example, /dev/binder can be accessed only in coredomain and /dev/vndbinder can be accessed only in non-coredomain.

Similar policies restrict the access to the shared libraries on system and vendor partitions. The following table shows the rights to access shared libraries of different categories:

Category	Partition	Accessible from coredomain	Accessible from non-coredomain
LL-NDK	System	Y	Y
LL-NDK-Private	System	Y	Y
VNDK-SP/VNDK-SP-Private	System	Y	Y
VNDK-SP-Ext	Vendor	Y	Y
VNDK	System	Y	Y
VNDK-Ext	Vendor	N	Y
FWK-ONLY	System	Y	N
FWK-ONLY-RS	System	Y	N
SP-HAL	Vendor	Y	Y
SP-HAL-Dep	Vendor	Y	Y
VND-ONLY	Vendor	N	Y

LL-NDK-Private and VNDK-SP-Private must be accessible from both domains because non-coredomain will indirectly access them. Similarly, SP-HAL-Dep must be accessible from coredomain because SP-HAL relies on it.

same_process_hal_file label
The following libraries exist in the vendor partition. Make these libraries accessible from both coredomain and non-coredomain.

VNDK-SP-Ext in /vendor/lib[64]/vndk-sp
SP-HAL in /vendor/lib[64] or /vendor/lib[64]/hw
SP-HAL-Dep in /vendor/lib[64] or /vendor/lib[64]/hw

Explicitly label these files as same_process_hal_file, because anything in vendor partition is by default not accessible to coredomain. Add lines similar to the following to the vendor-specific file_contexts file.

```
/vendor/lib(64)?/hw/libMySpHal\.so      u:object_r:same_process_hal_file:s0
/vendor/lib(64)?/vndk-sp/libBase\.so    u:object_r:same_process_hal_file:s0
/vendor/lib(64)?/libBaseInternal\.so    u:object_r:same_process_hal_file:s0
```

Great! Let's now explain this page in **simple words with examples** to help you understand the **directory layout**, **VNDK rules**, and **SEPolicy** in Android (especially from Android 8.0+).

✅ 1. Directory Layout (Who Keeps What?)

Think of Android like a **big city with two regions**:

- **System Region**: Managed by Android (Google/AOSP)
- **Vendor Region**: Managed by device manufacturer (OEM/SoC vendor)

Let's now see **who lives where**:

Folder	Who lives here	Examples
`/system/lib[64]`	All **framework libraries**	`libc.so`, `liblog.so`, `libbase.so`
`/system/lib[64]/vndk-sp`	VNDK-SP libraries (for same-process HALs)	`libutils.so`, `libhardware.so`
`/vendor/lib[64]`	Vendor-added libraries, HALs, extended VNDKs	`libcamera.so`, `libMySpHal.so`
`/vendor/lib[64]/vndk-sp`	Vendor-extended VNDK-SP	Vendor's own `libutils.so` etc.

👉 Think of `/system` as **the Android kitchen** and `/vendor` as **your mom's special cooking area** — each has its own tools, some are shared, but most must be used separately.

✅ 2. VNDK Rules in Simple Words

Why rules?

Because **framework and vendor** come from different companies and may **update separately**.

Let's see the rules like **traffic rules** between two areas:

🚦 Rule 1: Framework (system apps) must NOT use vendor libraries

🟥 **Forbidden**:

```sh
/system/bin/app → dlopen("/vendor/lib/libcamera.so") ❌
```

✅ **Allowed**:

```sh
/system/bin/app → /system/lib/libcamera_interface.so ✅
```

🚦 Rule 2: Vendor must NOT use random system libs

Vendor code **must only use**:

- LL-NDK libraries (always stable like `libc.so`)
- VNDK-SP (`libutils.so`, etc.)
- VNDK (`libbase.so`, etc.)

🟥 **Forbidden** (future releases):

```sh
/vendor/bin/hwservice → /system/lib/libfoo_private.so ❌
```

✅ **Allowed**:

```sh
/vendor/bin/hwservice → /system/lib/vndk-30/libutils.so ✅
```

🚦 Rule 3: No secret backdoors

No communication through:

- **`/data/vendor_shared_file`**
- **`sockets`**, **`pipes`**, custom **`binder`** without HIDL or AIDL

✅ Use **HIDL**, **AIDL**, or **HALs** to talk between framework and vendor.

🚦 Rule 4: Dependencies should be clean

For example:

- A SP-HAL (same process HAL) like **`libMySpHal.so`** must only depend on:
  - LL-NDK
  - VNDK-SP
  - Other SP-HAL libs

🟥 **Can't depend on**: **`libframework.so`**

✅ 3. SEPolicy Rules in Simple Words

SELinux separates **coredomain** (framework) and **non-coredomain** (vendor) for security.

Examples:

Resource	Who Can Access?
`/dev/binder`	Only system apps (coredomain)
`/dev/vndbinder`	Only vendor apps (non-coredomain)

📚 Library Access Table Summary

Library Type	Stored In	System (coredomain) can access	Vendor (non-coredomain) can access
LL-NDK	/system	✅	✅
VNDK-SP	/system	✅	✅
VNDK-SP-Ext	/vendor	✅	✅
VNDK-Ext	/vendor	❌	✅
Framework-only (FWK-ONLY)	/system	✅	❌
SP-HAL	/vendor	✅	✅
SP-HAL-Dep	/vendor	✅	✅
VND-ONLY	/vendor	❌	✅

✅ 4. same_process_hal_file Labeling

Because **/vendor/lib[64]** is **not accessible** to system processes **by default**, you have to **explicitly allow** system to access **SP-HALs** and **VNDK-SP-Ext**.

Example file_contexts entry:

```bash
/vendor/lib64/hw/libMySpHal\.so         u:object_r:same_process_hal_file:s0
/vendor/lib64/vndk-sp/libBase\.so       u:object_r:same_process_hal_file:s0
/vendor/lib64/libBaseInternal\.so       u:object_r:same_process_hal_file:s0
```

This tells SELinux:
✅ "Yes, system (coredomain) is allowed to load these vendor `.so` files."

🧠 Quick Recap in Real Words

- 🏢 **/system** is Android's official building; `/vendor` is OEM's own space.
- 🧱 Vendor code must only depend on **safe, stable system libraries (VNDK, LL-NDK)**.
- 🚓 There are **traffic rules** (VNDK rules) to make updates safe.
- 🛡️ SELinux protects system and vendor from each other.
- 🔒 You must **label vendor libraries** (like SP-HALs) if you want system to access them.

Would you like a **diagram** or a **real example from AOSP** showing how a HAL follows these rules?

13. RenderScript

bookmark_border
Caution: Starting with Android 12, the RenderScript APIs are deprecated. They will continue to function, but AOSP expects that device and component manufacturers will stop providing hardware-acceleration support over time. To take full advantage of GPU acceleration, AOSP recommends that you Migrate from RenderScript.
RenderScript is a framework for running computationally intensive tasks at high performance on Android. It is designed for use with data-parallel computation, although serial workloads can benefit as well. The RenderScript runtime parallelizes work across processors available on a device, such as multi-core CPUs and GPUs, enabling developers to focus on expressing algorithms rather than scheduling work. RenderScript is especially useful for apps performing image processing, computational photography, or computer vision.

Devices running Android 8.0 and higher use the following RenderScript framework and vendor HALs:

Figure 1. Vendor code linking to internal libs.

Differences from RenderScript in Android 7.x and lower include:

Two instances of RenderScript internal libs in a process. One set is for CPU fallback path and is from directly at /system/lib; the other set is for GPU path and is from /system/lib/vndk-sp.
RS internal libs in /system/lib are built as part of the platform and are updated as system.img is upgraded. However, libs in /system/lib/vndk-sp are built for the vendor and are not updated when system.img is upgraded (while they can be updated for a security fix, their ABI remains the same).
Vendor code (RS HAL, RS driver, and the bcc plugin) are linked against the RenderScript internal libs located at /system/lib/vndk-sp. They cannot link against libs in /system/lib because libs in that directory are built for the platform and thus may not be compatible with the vendor code (i.e., symbols may be removed). Doing so would make a framework-only OTA impossible.
Design
The following sections detail RenderScript design in Android 8.0 and higher.

RenderScript libs available to vendors
This section lists the RenderScript libs (known as Vendor NDK for Same-Process HALs or VNDK-SP) that are available to vendor code and which can be linked against. It also details additional libraries that are unrelated to RenderScript but which are also provided to vendor code.

While the following list of libraries might differ between Android releases, it is immutable for a specific Android release; for an up-to-date list of available libraries, refer to /system/etc/ld.config.txt.

Note: Libraries not listed below cannot be used by any vendor code; i.e. libLLVM.so cannot be used by the vendor's bcc plugin as the lib is not in the list.
RenderScript Libs	Non-RenderScript Libs
android.hardware.graphics.renderscript@1.0.so
libRS_internal.so
libRSCpuRef.so
libblas.so
libbcinfo.so
libcompiler_rt.so
libRSDriver.so
libc.so
libm.so
libdl.so
libstdc++.so
liblog.so
libnativewindow.so
libsync.so
libvndksupport.so
libbase.so
libc++.so
libcutils.so
libutils.so
libhardware.so
libhidlbase.so
libhidltransport.so
libhwbinder.so
liblzma.so

libz.so
libEGL.so
libGLESv1_CM.so
libGLESv2.so
Linker namespace configuration
The linking restriction that prevents libs not in VNDK-SP from being used by vendor code is enforced at runtime using the linker namespace. (For details, refer to the VNDK Design presentation.)

On a device running Android 8.0 and higher, all Same-Process HALs (SP-HALs) except RenderScript are loaded inside the linker namespace sphal. RenderScript is loaded into the RenderScript-specific namespace rs, a location that enables a slightly looser enforcement for RenderScript libs. Because the RS implementation needs to load the compiled bitcode, /data/*/*.so is added to the path of the rs namespace (other SP-HALs are not allowed to load libs from the data partition).

In addition, the rs namespace allows more libs than provided for by other namespaces. libmediandk.so and libft2.so are exposed to the rs namespace because libRS_internal.so has an internal dependency to these libraries.

Figure 2. Namespace configuration for linker.

Load drivers
CPU fallback path
Depending on the existence of the RS_CONTEXT_LOW_LATENCY bit when creating an RS context, either the CPU or GPU path is selected. When the CPU path is selected, libRS_internal.so (the main implementation of the RS framework) is directly dlopened from the default linker namespace where the platform version of RS libs are provided.

The RS HAL implementation from the vendor is not used at all when the CPU fallback path is taken, and an RsContext object is created with null mVendorDriverName. libRSDriver.so is (by default) dlopened and the driver lib is loaded from the default namespace because the caller (libRS_internal.so) is also loaded in the default namespace.

Figure 3. CPU fallback path.

GPU path
For the GPU path, the libRS_internal.so is loaded differently. First, libRS.so uses android.hardware.renderscript@1.0.so (and its underlying libhidltransport.so) to load android.hardware.renderscript@1.0-impl.so (a vendor implementation of RS HAL) into a different linker namespace called sphal. The RS HAL then dlopens libRS_internal.so in a another linker namespace called rs.

Vendors can provide their own RS driver by setting the build time flag OVERRIDE_RS_DRIVER, which is embedded into the RS HAL implementation (hardware/interfaces/renderscript/1.0/default/Context.cpp). This driver name is then dlopened for the RS context for the GPU path.

The creation of the RsContext object is delegated to the RS HAL implementation. The HAL calls back to the RS framework using rsContextCreateVendor() function with the name of the driver to use as an argument. The RS framework then loads the specified driver when the RsContext is initialized. In this case, the driver library is loaded into the rs namespace because the RsContext object is created inside the rs namespace and /vendor/lib is in the search path of the namespace.

Figure 4. GPU fallback path.

When transitioning from the default namespace to the sphal namespace, libhidltransport.so uses the android_load_sphal_library() function to explicitly order the dynamic linker to load the -impl.so library from the sphal namespace.

When transitioning from the sphal namespace to the rs namespace, loading is done indirectly by the following line in /system/etc/ld.config.txt:

namespace.sphal.link.rs.shared_libs = libRS_internal.so
This line specifies the dynamic linker should load libRS_internal.so from the rs namespace when the lib can't be found/loaded from the sphal namespace (which is always the case because sphal namespace does not search /system/lib/vndk-sp where libRS_internal.so resides). With this configuration, a simple dlopen() call to libRS_internal.so is enough to make the namespace transition.

Load bcc plugin
bcc plugin is a vendor-provided library loaded into the bcc compiler. Because bcc is a system process in the /system/bin directory, the bcc plugin library can be considered an SP-HAL (i.e., a vendor HAL that can be directly loaded into the system process without being binderized). As an SP-HAL, the bcc-plugin library:

Cannot link against framework-only libraries such as libLLVM.so.
Can link against only the VNDK-SP libraries available to the vendor.

This restriction is enforced by loading the bcc plugin into the sphal namespace using the android_sphal_load_library() function. In previous versions of Android, the plugin name was specified using the -load option and the lib was loaded using the simple dlopen() by libLLVM.so. In Android 8.0 and higher, this is specified in the -plugin option and the lib is directly loaded by the bcc itself. This option enables a non-Android-specific path to the open source LLVM project.

Figure 5. Loading bcc plugin, Android 7.x and lower.

Figure 6. Loading bcc plugin, Android 8.0 and higher.

Search paths for ld.mc
When executing ld.mc, some RS runtime libs are given as inputs to the linker. The RS bitcode from the app is linked against the runtime libs and when the converted bitcode is loaded into an app process, the runtime libs are again dynamically linked from the converted bitcode.

Runtime libs include:

libcompiler_rt.so
libm.so
libc.so
RS driver (either libRSDriver.so or OVERRIDE_RS_DRIVER)
When loading the compiled bitcode into the app process, provide the exact same library that was used by ld.mc. Otherwise, the compiled bitcode may not find a symbol which was available when it was linked.

To do so, RS framework uses different search paths for the runtime libs when executing ld.mc, depending on whether the RS framework itself is loaded from /system/lib or from /system/lib/vndk-sp. This can be determined by reading the address of an arbitrary symbol of a RS framework lib and using dladdr() to get the file path mapped to the address.

SELinux policy
As a result of the SELinux policy changes in Android 8.0 and higher, you must follow specific rules (enforced through neverallows) when labelling additional files in vendor partition:

vendor_file must be the default label in for all files in vendor partition. The platform policy requires this to access passthrough HAL implementations.
All new exec_types added in vendor partition through vendor SEPolicy must have vendor_file_type attribute. This is enforced through neverallows.
To avoid conflicts with future platform/framework updates, avoid labelling files other than exec_types in vendor partition.
All library dependencies for AOSP-identified same process HALs must be labelled as same_process_hal_file.
For details on SELinux policy, see Security-Enhanced Linux in Android.

ABI compatibility for bitcode
If no new APIs are added, which means no HAL version bump, the RS frameworks will keep using the existing GPU (HAL 1.0) driver.

For minor HAL changes (HAL 1.1) not affecting bitcode, the frameworks should fallback to CPU for these newly added APIs and keep using GPU (HAL 1.0) driver elsewhere.

For major HAL changes (HAL 2.0) affecting bitcode compilation/linking, RS frameworks should choose not to load vendor-provided GPU drivers and instead use the CPU or Vulkan path for acceleration.

Consuming RenderScript bitcode occurs in three stages:

Stage	Details
Compile
The input bitcode (.bc) for bcc must be in LLVM 3.2 bitcode format and bcc must be backward compatible with existing (legacy) apps.
However, the meta-data in .bc could change (there could new runtime functions, e.g., Allocation setters ∓ getters, math functions, etc.). Part of the runtime functions lives in libclcore.bc, part of them lives in LibRSDriver or vendor equivalent.
New runtime functions or breaking meta-data changes require incrementing the bitcode API level. Because vendor drivers won't be able to consume it, the HAL version must also be incremented.
Vendors may have their own compilers, but the conclusions/requirements for bcc also apply to those compilers.
Link
The compiled .o will be linked with vendor driver, e.g., libRSDriver_foo.so, and libcompiler_rt.so. The CPU path will link with libRSDriver.so.
If the .o requires a new runtime API from libRSDriver_foo, the vendor driver has to be updated to support it.
Certain vendors may have their own linkers, but the argument for ld.mc also apply to them.
Load
libRSCpuRef loads the shared object. If there are changes to this interface, a HAL version bump is needed.

Vendors would either rely on libRSCpuRef to load the shared object, or implement their own.
In addition to the HAL, runtime APIs and the exported symbols are also interfaces. Neither interface has changed since Android 7.0 (API 24) and there are no immediate plans to change it in Android 8.0 and beyond. However, if the interface does change, the HAL version will also increment.

Vendor implementations
Android 8.0 and higher requires some GPU driver changes for the GPU driver to work correctly.

Driver modules
Driver modules must not depend on any system libraries that are not in the list.
Driver must provide its own android.hardware.renderscript@1.0-impl_{NAME}, or declare the default implementation android.hardware.renderscript@1.0-impl as its dependency.
CPU implementation libRSDriver.so is a good example of how to remove non-VNDK-SP dependencies.
Bitcode compiler
You can compile RenderScript bitcode for the vendor driver in two ways:

Invoke vendor-specific RenderScript compiler in /vendor/bin/ (preferred method of GPU compilation). Similar to other driver modules, the vendor compiler binary cannot depend on any system library that is not in the list of RenderScript libs available to vendors.
Invoke system bcc: /system/bin/bcc with a vendor-provided bcc plugin; this plugin cannot depend on any system library that is not in the list of RenderScript libs available to vendors.
If the vendor bcc plugin needs to interfere with the CPU compilation and its dependency on libLLVM.so cannot be easily removed, the vendor should copy bcc (and all the non-LL-NDK dependencies, including libLLVM.so, libbcc.so) into /vendor partition.

In addition, vendors need to make the following changes:

Figure 7. Changes to vendor driver.

Copy libclcore.bc to /vendor partition. This ensures libclcore.bc, libLLVM.so, and libbcc.so are in sync.
Change the path to the bcc executable by setting RsdCpuScriptImpl::BCC_EXE_PATH from the RS HAL implementation.
Note: Restrictions for /vendor/bin/* processes are not fully implemented. While not recommended, it is theoretically possible to just copy bcc to /vendor/bin/ without copying its dependencies.
SELinux policy
SELinux policy affects both the driver and the compiler executables. All driver modules must be labeled same_process_hal_file in the device's file_contexts. For example:

/vendor/lib(64)?/libRSDriver_EXAMPLE\.so   u:object_r:same_process_hal_file:s0
The compiler executable must be able to be invoked by an app process, as does the vendor copy of bcc (/vendor/bin/bcc). For example:

device/vendor_foo/device_bar/sepolicy/file_contexts:
/vendor/bin/bcc   u:object_r:same_process_hal_file:s0
Legacy devices
Legacy devices are those that satisfy the following conditions:

PRODUCT_SHIPPING_API_LEVEL is lower than 26.
PRODUCT_FULL_TREBLE_OVERRIDE is not defined.
For legacy devices, the restrictions are not enforced when upgrading to Android 8.0 and higher, meaning the drivers can continue to link to libraries in /system/lib[64]. However, because of the architecture change related to OVERRIDE_RS_DRIVER, android.hardware.renderscript@1.0-impl must be installed to /vendor partition; failing to do so forces RenderScript runtime fallback to CPU path.

For information about the motivation for the Renderscript deprecation, see the Android Developers Blog: Android GPU Compute Going Forward. Resource information for this deprecation includes the following:

Migrate from Renderscript
RenderScriptMigration Sample
Intrinsics Replacement Toolkit README
Intrinsics ReplacementToolkit.kt

Let's break down this **very complex RenderScript documentation** into **simple words with examples**, so you fully understand how it works, how it changed in Android 8.0+, and why it's deprecated now.

🧠 What is RenderScript? (In Simple Words)

RenderScript was a framework in Android that helped you do **heavy math/computation** like:

- Image filters
- Computer vision
- Computational photography (e.g. beautify, night mode)

💡 It can use **CPU or GPU** automatically to speed things up.

🚨 Why is RenderScript Deprecated?

Starting in **Android 12**, Google says:

> "Stop using RenderScript. It's old. Use **GPU compute with Vulkan** or **OpenGL compute shaders** instead."

Because:

- New devices don't guarantee RenderScript GPU support.
- Google stopped improving it.
- Vulkan is faster, modern, and portable.

⚙️ Key Changes Since Android 8.0 (Treble-related)

Before Android 8.0, system and vendor code could mix freely.

But Android 8.0 introduced **Project Treble**, which separates:

- `/system` → Google's part (framework)
- `/vendor` → Manufacturer's part (drivers, HALs)

Now, RenderScript HAL (GPU drivers) must follow strict **library boundaries**.

🧱 Important RenderScript Files

Library	Purpose
`libRS_internal.so`	Main RenderScript engine
`libRSCpuRef.so`	CPU fallback driver
`libRSDriver.so`	GPU driver (default)
`android.hardware.graphics.renderscript@1.0.so`	HIDL interface for RenderScript HAL
`libcompiler_rt.so`, `libbcinfo.so`, etc.	Helper libs for RS compilation
`libclcore.bc`	RenderScript runtime code (bitcode)

🔗 Linking & Namespaces (Simple Analogy)

Think of **linker namespaces** like **buckets with allowed libraries**.

There are 3 main buckets (namespaces):

Namespace	Purpose
`default`	For framework/system code
`sphal`	For Same-Process HALs (vendor code)
`rs`	Special namespace for RenderScript HALs

🚫 Vendor code **cannot** access random system libraries like `libLLVM.so`.
✅ It can only link to allowed libraries from `VNDK-SP` bucket.

🧭 Paths and Libraries

Situation	Lib Loaded From
CPU fallback	`/system/lib/libRS_internal.so` (default namespace)
GPU path	`/system/lib/vndk-sp/libRS_internal.so` (rs namespace)
Vendor driver (GPU)	`/vendor/lib/libRSDriver_VENDOR.so`
Vendor compiler	`/vendor/bin/bcc`

🔁 RenderScript Execution Flow (Simple Words)

Let's say you run a RenderScript computation.

🚧 Stage 1: Compile `.rs` → `.bc` (bitcode) using `bcc`

- Bitcode is like a raw set of instructions
- Compiled using either system `/system/bin/bcc` or vendor `/vendor/bin/bcc`

🧱 Stage 2: Link `.bc` → `.so`

- The bitcode is **linked** with `libRSDriver.so` (GPU driver) or `libRSCpuRef.so` (CPU driver)

⚡ Stage 3: Load `.so` into App

- Runtime loads the `.so` into your app process
- Must be loaded with **exact same libraries** used at link time

---

🤖 RenderScript in Action: CPU vs GPU

Path	What Happens
CPU Path	RS engine uses `libRSCpuRef.so`, skips vendor HAL
GPU Path	RS engine loads vendor HAL → driver → GPU runs compute

Example:

```cpp
if (RS_CONTEXT_LOW_LATENCY flag) {
   // Use GPU → via HAL
} else {
   // Use CPU fallback
}
```

🛡️ SELinux Policy (Security Rules)

To protect the system:

1. **Vendor drivers** must be labeled `same_process_hal_file`
   Example:

   ```bash
   /vendor/lib64/libRSDriver_EXAMPLE\.so  u:object_r:same_process_hal_file:s0
   ```

2. **Vendor bcc compiler** (if copied) must be labeled too:

   ```bash
   /vendor/bin/bcc  u:object_r:same_process_hal_file:s0
   ```

💾 Legacy Devices

Devices launched **before Android 8.0**:

- Don't need to follow Treble rules strictly
- Can still link to `/system/lib` directly
- But must move HAL to `/vendor` to work properly with newer RS runtime

✅ Summary Table (For Quick Understanding)

Component	Location	Used By	Example
RS Engine	`/system/lib/libRS_internal.so`	CPU path	System
RS GPU HAL	`/vendor/lib/libRSDriver_foo.so`	GPU path	Vendor
bcc compiler	`/system/bin/bcc` or `/vendor/bin/bcc`	Compilation	Vendor
Namespace	`default`, `sphal`, `rs`	Runtime linker	Enforces VNDK rules
SELinux	`same_process_hal_file`	Grants access to vendor `.so` from system	Labels required

🔄 Migration Path (Post-Android 12)

Since RenderScript is deprecated:

✅ Use **Vulkan Compute Shaders** or **OpenGL Compute**
✅ Use **Neural Networks API (NNAPI)** for ML workloads
✅ Use **Intrinsics Replacement Toolkit** for image filters

🚀 Want an Example Migration?

Let me know if you want:

- 🧑‍💻 A simple RenderScript image filter
- 🚀 Its migration to Vulkan compute
- 📏 Benchmarking difference between RS and Vulkan

I'll walk you through it.

Would you like that?

14. can you explain simple words with example and we are exploring renderscript here

Sure! Let me explain **RenderScript** in simple words, with examples, and how it works in Android.

🌟 What is RenderScript?

RenderScript is a special programming framework in Android for writing **high-performance code**, especially useful for:

- **Image processing**
- **Computer vision**
- **Heavy mathematical operations**

You write your algorithm once, and Android will **automatically run it using all available CPU cores or GPU** for maximum performance.

🧠 How RenderScript Works (Simple Flow)

Let's say you want to **blur an image**.

🔸 Without RenderScript:

You write a Java/Kotlin loop to go pixel by pixel. It's **slow**, uses just one CPU core.

🔸 With RenderScript:

You write your logic using RenderScript's C-like language. It runs in **parallel** on:

- All CPU cores
- Or even on the **GPU** (if available)

This gives **huge speedup**.

🏗️ RenderScript Components

Example: Blur an image using RenderScript (simplified)

```kotlin
val input = Allocation.createFromBitmap(rs, bitmap)
val output = Allocation.createTyped(rs, input.type)
val script = ScriptIntrinsicBlur.create(rs, Element.U8_4(rs))

script.setRadius(10f)
script.setInput(input)
script.forEach(output)

output.copyTo(outputBitmap)
```

Here:

- **`Allocation`** → Input/output memory
- **`ScriptIntrinsicBlur`** → Built-in RenderScript blur logic
- **`setRadius`** → How strong the blur should be
- **`forEach()`** → Run it on every pixel **in parallel**

🧱 RenderScript System Design (Android 8.0+)

From Android 8 onward, the system divides the RenderScript libraries like this:

Where	What
`/system/lib`	Platform version (used by framework / system)

Where	What
`/system/lib/vndk-sp`	**Stable** version for **vendor drivers**
`/vendor/lib`	Drivers (e.g., GPU driver for RS), RS HAL

This is done because:

- **System updates should not break vendor drivers**
- So, vendor drivers use stable RS libraries from `vndk-sp`, not from `/system/lib`

📂 Example Directory Layout

- `/system/lib/libRS_internal.so` → Main RenderScript library for CPU
- `/system/lib/vndk-sp/libRS_internal.so` → For **vendor GPU drivers**
- `/vendor/lib/libRSDriver_MYGPU.so` → Vendor GPU driver

💡 Real-World Example: GPU Blur on Samsung Device

Let's say you're using a Samsung phone with a Mali GPU. Here's what happens:

1. App uses RenderScript to blur an image.
2. Android checks if GPU driver is available.
3. If yes, it uses `/vendor/lib/libRSDriver_Mali.so` (provided by Samsung).
4. This driver links to `/system/lib/vndk-sp/libRS_internal.so` to access core RenderScript code.

That way:

- Samsung can update their driver independently.
- Google can update Android without breaking Samsung's driver.

🛡️ SELinux Security Rules

Because Android is secure, not all libraries are visible to each other. Special labels are needed:

```bash
/vendor/lib/libRSDriver_MyGpu.so  → u:object_r:same_process_hal_file:s0
```

This tells the system: "Hey, this vendor driver is **safe** and can be loaded by system apps too."

⚠️ Important: RenderScript is Deprecated

Since **Android 12**, Google has deprecated RenderScript:

- It's still there for compatibility.
- But **no more hardware acceleration updates.**

Instead, Google recommends:
✅ Using **GPU compute via Vulkan or OpenGL ES**
✅ Using **RenderScript Intrinsics Replacement Toolkit** (Google's new path)

🔚 Summary

Term	Meaning
RenderScript	A framework to write parallel code on CPU/GPU
VNDK-SP	Set of libraries safe for both system and vendor
HAL	Hardware abstraction layer, here used for GPU
Namespace	Keeps system/vendor libraries separate but connected
Deprecated	Google suggests moving to Vulkan or other GPU compute APIs

Would you like to see a **step-by-step guide on writing a RenderScript kernel**, or how to migrate to a Vulkan-based GPU compute?

15. ABI stability

bookmark_border
Application Binary Interface (ABI) stability is a prerequisite of framework-only updates because vendor modules may depend on the Vendor Native Development Kit (VNDK) shared libraries that reside in the system partition. Within an Android release, newly-built VNDK shared libraries must be ABI-compatible to previously released VNDK shared libraries so vendor modules can work with those libraries without recompilation and without runtime errors. Between Android releases, VNDK libraries can be changed and there are no ABI guarantees.

To help ensure ABI compatibility, Android 9 includes a header ABI checker, as described in the following sections.

About VNDK and ABI compliance
The VNDK is a restrictive set of libraries that vendor modules may link to and which enable framework-only updates. ABI compliance refers to the ability of a newer version of a shared library to work as expected with a module that is dynamically linked to it (i.e. works as an older version of the library would).

About exported symbols
An exported symbol (also known as a global symbol) refers to a symbol that satisfies all of the following:

Exported by the public headers of a shared library.
Appears in the .dynsym table of the .so file corresponding to the shared library.
Has WEAK or GLOBAL binding.
Visibility is DEFAULT or PROTECTED.
Section index is not UNDEFINED.
Type is either FUNC or OBJECT.
The public headers of a shared library are defined as the headers available to other libraries/binaries through the export_include_dirs, export_header_lib_headers, export_static_lib_headers, export_shared_lib_headers, and export_generated_headers attributes in Android.bp definitions of the module corresponding to the shared library.

About reachable types
A reachable type is any C/C++ built-in or user-defined type that is reachable directly or indirectly through an exported symbol AND exported through public headers. For example, libfoo.so has function Foo, which is an exported symbol found in the .dynsym table. The libfoo.so library includes the following:

foo_exported.h	foo.private.h

```
typedef struct foo_private foo_private_t;

typedef struct foo {
 int m1;
 int *m2;
 foo_private_t *mPfoo;
} foo_t;

typedef struct bar {
 foo_t mfoo;
} bar_t;

bool Foo(int id, bar_t *bar_ptr);

typedef struct foo_private {
 int m1;
 float mbar;
} foo_private_t;
```

Android.bp

```
cc_library {
 name : libfoo,
 vendor_available: true,
 vndk {
  enabled : true,
 }
 srcs : ["src/*.cpp"],
 export_include_dirs : [
```

```
  "exported"
 ],
}
```

.dynsym table

Num	Value	Size	Type	Bind	Vis	Ndx	Name
1	0	0	FUNC	GLOB	DEF	UND	dlerror@libc
2	1ce0	20	FUNC	GLOB	DEF	12	Foo

Looking at Foo, direct/indirect reachable types include:

Type	Description
bool	Return type of Foo.
int	Type of first Foo parameter.
bar_t *	Type of second Foo parameter. By way of bar_t *, bar_t is exported through foo_exported.h.

bar_t contains a member mfoo, of type foo_t, which is exported through foo_exported.h, which results in more types being exported:
int : is the type of m1.
int * : is the type of m2.
foo_private_t * : is the type of mPfoo.

However, foo_private_t is NOT reachable because it isn't exported through foo_exported.h. (foo_private_t * is opaque, therefore changes made to foo_private_t are allowed.)
A similar explanation can be given for types reachable through base class specifiers and template parameters as well.

Ensure ABI compliance
ABI compliance must be ensured for the libraries marked vendor_available: true and vndk.enabled: true in the corresponding Android.bp files. For example:

```
cc_library {
  name: "libvndk_example",
  vendor_available: true,
  vndk: {
    enabled: true,
  }
}
```

For data types reachable directly or indirectly by an exported function, the following changes to a library are classified as ABI-breaking:

Data type	Description
Structures and Classes
Change the size of the class type or the struct type.
Base classes
Add or remove base classes.
Add or remove virtually inherited base classes.
Change the order of base classes.
Member functions
Remove member functions*.
Add or remove arguments from member functions.
Change the argument types or the return types of member functions*.
Change the virtual table layout.
Data members
Remove static data members.
Add or remove non-static data members.
Change the types of data members.
Change the offsets to non-static data members**.
Change the const, volatile, and/or restricted qualifiers of data members***.
Downgrade the access specifiers of data members***.
Change the template arguments.
Unions
Add or remove data members.
Change the size of the union type.
Change the types of data members.
Enumerations
Change the underlying type.
Change the names of enumerators.
Change the values of enumerators.
Global Symbols

Remove the symbols exported by public headers.
For global symbols of type FUNC
Add or remove arguments.
Change the argument types.
Change the return type.
Downgrade the access specifier***.
For global symbols of type OBJECT
Change the corresponding C/C++ type.
Downgrade the access specifier***.
* Both public and private member functions must not be changed or removed because public inline functions can refer to private member functions. Symbol references to private member functions can be kept in caller binaries. Changing or removing private member functions from shared libraries can result in backward-incompatible binaries.

** The offsets to public or private data members must not be changed because inline functions can refer to these data members in their function body. Changing data member offsets can result in backward-incompatible binaries.

*** While these don't change the memory layout of the type, there are semantic differences that could lead to libraries not functioning as expected.

Use ABI compliance tools
When a VNDK library is built, the library's ABI is compared with the corresponding ABI reference for the version of the VNDK being built. Reference ABI dumps are located in:

${ANDROID_BUILD_TOP}/prebuilts/abi-dumps/vndk/<PLATFORM_VNDK_VERSION>/<BINDER_BITNESS>/<ARCH>/source-based
For example, on building libfoo for x86 at API level 27, libfoo's inferred ABI is compared with its reference at:

${ANDROID_BUILD_TOP}/prebuilts/abi-dumps/vndk/27/64/x86/source-based/libfoo.so.lsdump
ABI breakage error
On ABI breakages, the build log displays warnings with the warning type and a path to the abi-diff report. For example, if libbinder's ABI has an incompatible change, the build system throws an error with a message similar to the following:

****
error: VNDK library: libbinder.so's ABI has INCOMPATIBLE CHANGES
Please check compatibility report at:
out/soong/.intermediates/frameworks/native/libs/binder/libbinder/android_arm64_armv8-a_cortex-a73_vendor_shared/libbinder.so.abidiff
****
---- Please update abi references by running
platform/development/vndk/tools/header-checker/utils/create_reference_dumps.py -l libbinder ----
Build VNDK library ABI checks
When a VNDK library is built:

header-abi-dumper processes the source files compiled to build the VNDK library (the library's own source files as well as source files inherited through static transitive dependencies), to produce .sdump files that correspond to each source.
sdump creation
Figure 1. Creating the .sdump files
header-abi-linker then processes the .sdump files (using either a version script provided to it or the .so file corresponding to the shared library) to produce a .lsdump file that logs all of the ABI information corresponding to the shared library.
lsdump creation
Figure 2. Creating the .lsdump file
header-abi-diff compares the .lsdump file with a reference .lsdump file to produce a diff report that outlines the differences in the ABIs of the two libraries.
abi diff creation
Figure 3. Creating the diff report
header-abi-dumper
The header-abi-dumper tool parses a C/C++ source file and dumps the ABI inferred from that source file into an intermediate file. The build system runs header-abi-dumper on all compiled source files while also building a library that includes the source files from transitive dependencies.

Inputs
A C/C++ source file
Exported include directories
Compiler flags
Output A file that describes the ABI of the source file (for example, foo.sdump represents foo.cpp's ABI).

Currently .sdump files are in JSON format, which isn't guaranteed to be stable across future releases. As such, .sdump file formatting should be considered a build system implementation detail.

For example, libfoo.so has the following source file foo.cpp:

```
#include <stdio.h>
#include <foo_exported.h>

bool Foo(int id, bar_t *bar_ptr) {
    if (id > 0 && bar_ptr->mfoo.m1 > 0) {
        return true;
    }
    return false;
}
```

You can use header-abi-dumper to generate an intermediate .sdump file that represents the ABI presented by the source file using:

```
$ header-abi-dumper foo.cpp -I exported -o foo.sdump -- -I exported -x c++
```

This command tells header-abi-dumper to parse foo.cpp with the compiler flags following --, and emit the ABI information that is exported by the public headers in the exported directory. The following is foo.sdump generated by header-abi-dumper:


```
{
 "array_types" : [],
 "builtin_types" :
 [
  {
   "alignment" : 4,
   "is_integral" : true,
   "linker_set_key" : "_ZTIi",
   "name" : "int",
   "referenced_type" : "_ZTIi",
   "self_type" : "_ZTIi",
   "size" : 4
  }
 ],
 "elf_functions" : [],
 "elf_objects" : [],
 "enum_types" : [],
 "function_types" : [],
 "functions" :
 [
  {
   "function_name" : "FooBad",
   "linker_set_key" : "_Z6FooBadiP3foo",
   "parameters" :
   [
    {
     "referenced_type" : "_ZTIi"
    },
    {
     "referenced_type" : "_ZTIP3foo"
    }
   ],
   "return_type" : "_ZTI3bar",
   "source_file" : "exported/foo_exported.h"
  }
 ],
 "global_vars" : [],
 "lvalue_reference_types" : [],
 "pointer_types" :
 [
  {
   "alignment" : 8,
   "linker_set_key" : "_ZTIP11foo_private",
```

```
   "name" : "foo_private *",
   "referenced_type" : "_ZTI11foo_private",
   "self_type" : "_ZTIP11foo_private",
   "size" : 8,
   "source_file" : "exported/foo_exported.h"
  },
  {
   "alignment" : 8,
   "linker_set_key" : "_ZTIP3foo",
   "name" : "foo *",
   "referenced_type" : "_ZTI3foo",
   "self_type" : "_ZTIP3foo",
   "size" : 8,
   "source_file" : "exported/foo_exported.h"
  },
  {
   "alignment" : 8,
   "linker_set_key" : "_ZTIPi",
   "name" : "int *",
   "referenced_type" : "_ZTIi",
   "self_type" : "_ZTIPi",
   "size" : 8,
   "source_file" : "exported/foo_exported.h"
  }
 ],
 "qualified_types" : [],
 "record_types" :
 [
  {
   "alignment" : 8,
   "fields" :
   [
    {
     "field_name" : "mfoo",
     "referenced_type" : "_ZTI3foo"
    }
   ],
   "linker_set_key" : "_ZTI3bar",
   "name" : "bar",
   "referenced_type" : "_ZTI3bar",
   "self_type" : "_ZTI3bar",
   "size" : 24,
   "source_file" : "exported/foo_exported.h"
  },
  {
   "alignment" : 8,
   "fields" :
   [
    {
     "field_name" : "m1",
     "referenced_type" : "_ZTIi"
    },
    {
     "field_name" : "m2",
     "field_offset" : 64,
     "referenced_type" : "_ZTIPi"
    },
    {
     "field_name" : "mPfoo",
     "field_offset" : 128,
     "referenced_type" : "_ZTIP11foo_private"
    }
   ],
   "linker_set_key" : "_ZTI3foo",
   "name" : "foo",
   "referenced_type" : "_ZTI3foo",
```

```
   "self_type" : "_ZTI3foo",
   "size" : 24,
   "source_file" : "exported/foo_exported.h"
  }
 ],
 "rvalue_reference_types" : []
}
```


foo.sdump contains ABI information exported by the source file foo.cpp and the public headers, for example,

record_types. Refer to structs, unions, or classes defined in the public headers. Each record type has information about its fields, its size, access specifier, the header file it's defined in, and other attributes.
pointer_types. Refer to pointer types directly/indirectly referenced by the exported records/functions in the public headers, along with the type the pointer points to (through the referenced_type field in type_info). Similar information is logged in the .sdump file for qualified types, built-in C/C++ types, array types, and lvalue and rvalue reference types. Such information allows recursive diffing.
functions. Represent functions exported by public headers. They also have information about the function's mangled name, the return type, the types of the parameters, the access specifier, and other attributes.
Tip: To get help with the header-abi-dumper tool, run header-abi-dumper --help.
header-abi-linker
The header-abi-linker tool takes the intermediate files produced by header-abi-dumper as input then links those files:

Inputs
Intermediate files produced by header-abi-dumper
Version script/Map file (optional)
.so file of the shared library
Exported include directories
Output   A file that describes the ABI of a shared library (for example, libfoo.so.lsdump represents libfoo's ABI).
The tool merges the type graphs in all the intermediate files given to it, taking into account one-definition (user-defined types in different translation units with the same fully qualified name, might be semantically different) differences across translation units. The tool then parses either a version script or the .dynsym table of the shared library (.so file) to make a list of the exported symbols.

For example, libfoo consists of foo.cpp and bar.cpp. header-abi-linker could be invoked to create the complete linked ABI dump of libfoo as follows:

```
header-abi-linker -I exported foo.sdump bar.sdump \
                -o libfoo.so.lsdump \
                -so libfoo.so \
                -arch arm64 -api current
```

Example command output in libfoo.so.lsdump:


```
{
 "array_types" : [],
 "builtin_types" :
 [
  {
   "alignment" : 1,
   "is_integral" : true,
   "is_unsigned" : true,
   "linker_set_key" : "_ZTIb",
   "name" : "bool",
   "referenced_type" : "_ZTIb",
   "self_type" : "_ZTIb",
   "size" : 1
  },
  {
   "alignment" : 4,
   "is_integral" : true,
   "linker_set_key" : "_ZTIi",
   "name" : "int",
   "referenced_type" : "_ZTIi",
   "self_type" : "_ZTIi",
   "size" : 4
  }
 ],
 "elf_functions" :
```

```
[
 {
  "name" : "_Z3FooiP3bar"
 },
 {
  "name" : "_Z6FooBadiP3foo"
 }
],
"elf_objects" : [],
"enum_types" : [],
"function_types" : [],
"functions" :
[
 {
  "function_name" : "Foo",
  "linker_set_key" : "_Z3FooiP3bar",
  "parameters" :
  [
   {
    "referenced_type" : "_ZTIi"
   },
   {
    "referenced_type" : "_ZTIP3bar"
   }
  ],
  "return_type" : "_ZTIb",
  "source_file" : "exported/foo_exported.h"
 },
 {
  "function_name" : "FooBad",
  "linker_set_key" : "_Z6FooBadiP3foo",
  "parameters" :
  [
   {
    "referenced_type" : "_ZTIi"
   },
   {
    "referenced_type" : "_ZTIP3foo"
   }
  ],
  "return_type" : "_ZTI3bar",
  "source_file" : "exported/foo_exported.h"
 }
],
"global_vars" : [],
"lvalue_reference_types" : [],
"pointer_types" :
[
 {
  "alignment" : 8,
  "linker_set_key" : "_ZTIP11foo_private",
  "name" : "foo_private *",
  "referenced_type" : "_ZTI11foo_private",
  "self_type" : "_ZTIP11foo_private",
  "size" : 8,
  "source_file" : "exported/foo_exported.h"
 },
 {
  "alignment" : 8,
  "linker_set_key" : "_ZTIP3bar",
  "name" : "bar *",
  "referenced_type" : "_ZTI3bar",
  "self_type" : "_ZTIP3bar",
  "size" : 8,
  "source_file" : "exported/foo_exported.h"
 },
```

```
  {
   "alignment" : 8,
   "linker_set_key" : "_ZTIP3foo",
   "name" : "foo *",
   "referenced_type" : "_ZTI3foo",
   "self_type" : "_ZTIP3foo",
   "size" : 8,
   "source_file" : "exported/foo_exported.h"
  },
  {
   "alignment" : 8,
   "linker_set_key" : "_ZTIPi",
   "name" : "int *",
   "referenced_type" : "_ZTIi",
   "self_type" : "_ZTIPi",
   "size" : 8,
   "source_file" : "exported/foo_exported.h"
  }
 ],
 "qualified_types" : [],
 "record_types" :
 [
  {
   "alignment" : 8,
   "fields" :
   [
    {
     "field_name" : "mfoo",
     "referenced_type" : "_ZTI3foo"
    }
   ],
   "linker_set_key" : "_ZTI3bar",
   "name" : "bar",
   "referenced_type" : "_ZTI3bar",
   "self_type" : "_ZTI3bar",
   "size" : 24,
   "source_file" : "exported/foo_exported.h"
  },
  {
   "alignment" : 8,
   "fields" :
   [
    {
     "field_name" : "m1",
     "referenced_type" : "_ZTIi"
    },
    {
     "field_name" : "m2",
     "field_offset" : 64,
     "referenced_type" : "_ZTIPi"
    },
    {
     "field_name" : "mPfoo",
     "field_offset" : 128,
     "referenced_type" : "_ZTIP11foo_private"
    }
   ],
   "linker_set_key" : "_ZTI3foo",
   "name" : "foo",
   "referenced_type" : "_ZTI3foo",
   "self_type" : "_ZTI3foo",
   "size" : 24,
   "source_file" : "exported/foo_exported.h"
  }
 ],
 "rvalue_reference_types" : []
```

}
The header-abi-linker tool:

Links the .sdump files provided to it (foo.sdump and bar.sdump), filtering out the ABI information not present in the headers residing in the directory: exported.
Parses libfoo.so, and collects information about the symbols exported by the library through its .dynsym table.
Adds _Z3FooiP3bar and _Z6FooBadiP3foo.
libfoo.so.lsdump is the final generated ABI dump of libfoo.so.

Tip: To get help with the header-abi-linker tool, run header-abi-linker --help.
header-abi-diff
The header-abi-diff tool compares two .lsdump files representing the ABI of two libraries and produces a diff report stating the differences between the two ABIs.

Inputs
.lsdump file representing the ABI of an old shared library.
.lsdump file representing the ABI of a new shared library.
Output A diff report stating the differences in the ABIs offered by the two shared libraries compared.
The ABI diff file is in protobuf text format. The format is subject to change in future releases.

For example, you have two versions of libfoo: libfoo_old.so and libfoo_new.so. In libfoo_new.so, in bar_t, you change the type of mfoo from foo_t to foo_t *. Since bar_t is a reachable type, this should be flagged as an ABI breaking change by header-abi-diff.

To run header-abi-diff:

```
header-abi-diff -old libfoo_old.so.lsdump \
        -new libfoo_new.so.lsdump \
        -arch arm64 \
        -o libfoo.so.abidiff \
        -lib libfoo
```

Example command output in libfoo.so.abidiff:

```
lib_name: "libfoo"
arch: "arm64"
record_type_diffs {
  name: "bar"
  type_stack: "Foo-> bar *->bar "
  type_info_diff {
    old_type_info {
      size: 24
      alignment: 8
    }
    new_type_info {
      size: 8
      alignment: 8
    }
  }
  fields_diff {
    old_field {
      referenced_type: "foo"
      field_offset: 0
      field_name: "mfoo"
      access: public_access
    }
    new_field {
      referenced_type: "foo *"
      field_offset: 0
      field_name: "mfoo"
      access: public_access
    }
  }
}
```

The libfoo.so.abidiff contains a report of all ABI breaking changes in libfoo. The record_type_diffs message indicates a record has changed and lists the incompatible changes, which include:

The size of the record changing from 24 bytes to 8 bytes.
The field type of mfoo changing from foo to foo * (all typedefs are stripped off).
The type_stack field indicates how header-abi-diff reached the type that changed (bar). This field may be interpreted as Foo is an exported function that takes in bar * as parameter, that points to bar, which was exported and changed.

Tip: To get help with the header-abi-diff tool, run header-abi-diff --help. You can also refer to development/vndk/tools/header-checker/README.md.
Enforce ABI and API
To enforce the ABI and API of VNDK shared libraries, ABI references must be checked into ${ANDROID_BUILD_TOP}/prebuilts/abi-dumps/vndk/.
To create these references, run the following command:

${ANDROID_BUILD_TOP}/development/vndk/tools/header-checker/utils/create_reference_dumps.py
After creating the references, any change made to the source code that results in an incompatible ABI/API change in a VNDK library now results in a build error.

To update ABI references for specific libraries, run the following command:

${ANDROID_BUILD_TOP}/development/vndk/tools/header-checker/utils/create_reference_dumps.py -l <lib1> -l <lib2>
For example, to update libbinder ABI references, run:

${ANDROID_BUILD_TOP}/development/vndk/tools/header-checker/utils/create_reference_dumps.py -l libbin

Let's break down **ABI stability in Android** and tools like `header-abi-dumper`, `header-abi-linker`, and `header-abi-diff` into **simple words with examples**, especially for a kernel or system developer like you.

🧠 What is ABI?

ABI = Application Binary Interface

🔹 It defines **how compiled code (binaries)** can interact with each other:

- Function signatures
- Return types
- Struct layouts
- Symbol names
- Calling conventions

If the **ABI breaks**, then **old vendor drivers/modules** may **crash or behave wrongly**, even if they were compiled correctly in the past.

✅ ABI Stability in Android

🔹 Why does Android care about ABI stability?

Android uses **framework-only updates**:

- System partition (AOSP framework + VNDK) is updated by Google/OEMs.
- **Vendor partition (drivers, HALs)** must still work without recompiling!

  So if a VNDK library like `libbinder.so` changes in a way that breaks ABI, **your vendor driver using it may crash.**

🧱 What is VNDK?

VNDK = Vendor Native Development Kit

It's a **set of libraries** from the system partition that vendor HALs can link to **safely**.

Examples:

- `libc++`
- `libutils`
- `libbinder`
- `liblog`

They must be **ABI-stable within a release**.

🛠️ How does Android enforce ABI stability?

Google added **tools** to detect **incompatible ABI changes**:

1. **`header-abi-dumper`** → Extract ABI info from each **`.cpp`** file.
2. **`header-abi-linker`** → Link all ABI info into **`.lsdump`** file.
3. **`header-abi-diff`** → Compare **`.lsdump`** files to detect **breakage**.

📦 Simple Example

Imagine you have a VNDK library:

```c++
// foo_exported.h

struct foo {
  int m1;
  int* m2;
};

bool Foo(int id, foo* ptr);
```

And your **`Android.bp`**:

```python
cc_library {
  name: "libfoo",
  vendor_available: true,
  vndk: {
    enabled: true,
  },
  export_include_dirs: ["exported"],
}
```

You compile **`libfoo.so`**.

🧪 ABI Dump Tools in Action

🔹 Step 1: Dump ABI from `foo.cpp`

```bash
header-abi-dumper foo.cpp -I exported -o foo.sdump -- -x c++
```

It reads the public headers and dumps types, functions, pointers, etc.

🔹 Step 2: Link all ABI dumps

```bash
header-abi-linker -I exported foo.sdump -o libfoo.so.lsdump -so libfoo.so -arch arm64 -api current
```

Now you get **`libfoo.so.lsdump`** — **the full ABI state** of your library.

🔹 Step 3: Compare old vs new

If someone changes **foo** struct like this:

```c++
struct foo {
  float m1;  // ⚠ changed from int to float!
  int* m2;
};
```

You do:

```bash
header-abi-diff -old libfoo_old.so.lsdump \
                -new libfoo_new.so.lsdump \
                -arch arm64 \
                -o libfoo.abidiff \
                -lib libfoo
```

💥 Output: ABI break detected! (because **int** → **float** is incompatible)

🚨 What is an ABI-breaking change?

Examples:

Type	Breaking Change
struct	Add/remove fields
class	Add base class
function	Change return type
global function	Change args
pointer	Change from **foo** to **foo***
layout	Change offset of struct members

🔍 What's considered exported?

Only:

- Public headers listed in **export_include_dirs**
- Symbols visible in **.dynsym**
- Global/weak symbols with **DEFAULT/PROTECTED** visibility

💡 What are .sdump and .lsdump files?

File	What it means
.sdump	ABI info from **one .cpp file**
.lsdump	Linked ABI info from **whole library**
.abidiff	**Difference** between old and new **.lsdump**

These files are in **JSON** or **protobuf text** format (not stable for external use).

🗂️ Where are reference ABI dumps?

When you build Android, reference ABI dumps are stored in:

```bash
prebuilts/abi-dumps/vndk/<version>/<arch>/source-based/libfoo.so.lsdump
```

You compare current **.lsdump** to this to detect breakage.

🧰 Fixing ABI errors during build

You might see this error in Soong:

```markdown
*****************************************************
error: VNDK library: libbinder.so's ABI has INCOMPATIBLE CHANGES
Please check compatibility report at:
out/soong/…/libbinder.so.abidiff
*****************************************************
```

If the change is **intentional and safe**, update reference like this:

```bash
development/vndk/tools/header-checker/utils/create_reference_dumps.py -l libbinder
```

This updates the ABI `.lsdump` reference.

🔐 Summary (Plain English)

Concept	Explanation
ABI	Defines how binaries (e.g., system + vendor) talk
ABI breaking change	Change that causes old vendor code to crash
VNDK	Set of system libraries vendor code can use
ABI tools	Catch accidental ABI breaks at build time
`.sdump`/`.lsdump`	Represent your library's ABI
`.abidiff`	Shows what's broken between old and new

Would you like a **real-world demo with example structs and outputs**, or help using the ABI tools on your AOSP build?

16. Prebuilt ABI usages checker

bookmark_border
Android shared libraries evolve from time to time. Keeping prebuilt binaries up-to-date requires considerable effort. In Android 9 or earlier, the prebuilt binaries that depend on removed libraries or ABIs only fail to link at run-time. Developers have to trace the logs to find the outdated prebuilt binaries. In Android 10, a symbol-based ABI usages checker is introduced. The checker can detect outdated prebuilt binaries at build-time, so that shared library developers can know which prebuilt binaries might be broken by their change and which prebuilt binaries must be re-built.

Symbol-based ABI usages checker
The symbol-based ABI usages checker emulates the Android dynamic linker on host. The checker links the prebuilt binary with the dependencies of the prebuilt binary and checks whether all undefined symbols are resolved.

First, the checker checks the target architecture of the prebuilt binary. If the prebuilt binary does not target ARM, AArch64, x86, or x86-64 architecture, the checker skips the prebuilt binary.

Second, the dependencies of the prebuilt binary must be listed in LOCAL_SHARED_LIBRARIES or shared_libs. The build system resolves the module names to the matching variant (i.e. core vs. vendor) of the shared libraries.

Third, the checker compares the DT_NEEDED entries to LOCAL_SHARED_LIBRARIES or shared_libs. In particular, the checker extracts the DT_SONAME entry from each shared libraries and compares these DT_SONAME with the DT_NEEDED entries recorded in the prebuilt binary. If there is a mismatch, an error message is emitted.

Fourth, the checker resolves the undefined symbols in the prebuilt binary. Those undefined symbols must be defined in one of the dependencies and the symbol binding must be either GLOBAL or WEAK. If an undefined symbol cannot be resolved, an error message is emitted.

Prebuilts module properties
Dependencies of the prebuilt binary must be specified in one of the following:

Android.bp: shared_libs: ["libc", "libdl", "libm"],
Android.mk: LOCAL_SHARED_LIBRARIES := libc libdl libm
If the prebuilt binary is designed to have some unresolvable undefined symbols, specify one of the following:

Android.bp: allow_undefined_symbols: true,
Android.mk: LOCAL_ALLOW_UNDEFINED_SYMBOLS := true
To have the prebuilt binary skip the ELF file check, specify one of the following:

Android.bp: check_elf_files: false,
Android.mk: LOCAL_CHECK_ELF_FILES := false
Run the checker
The checker covers all ELF prebuilt modules during the Android build process.

To run the checker alone for faster turnaround times:

m check-elf-files
ABI error fixer
The automatic fixer can help resolve ABI check errors. Simply run the fixer with the Android.bp / Android.mk as input, and the fixer would print the suggested fix to stdout. Optionally, run the fixer with the --in-place option to directly update the Android.bp / Android.mk with the suggested fix.

For Android.bp,

m fix_android_bp_prebuilt
Print the fixed Android.bp to stdout.
fix_android_bp_prebuilt <path-to-Android.bp>
Update the Android.bp in place.
fix_android_bp_prebuilt --in-place <path-to-Android.bp>
For Android.mk,

m fix_android_mk_prebuilt

```
# Print the fixed Android.mk to stdout.
fix_android_mk_prebuilt <path-to-Android.mk>
# Update the Android.mk in place.
fix_android_mk_prebuilt --in-place <path-to-Android.mk>
```

Let's explain the **Prebuilt ABI Usages Checker** in **simple words with examples**, especially relevant if you're integrating or maintaining prebuilt `.so` libraries in AOSP (e.g., vendor drivers, HALs, RenderScript backends, etc.).

🧠 What's the Problem?

When you use **prebuilt `.so` libraries** (not compiled from source), these libraries depend on other system libraries like `libc.so`, `libm.so`, or `liblog.so`.

👉 If the **system library ABI changes** (e.g., a function is removed), then your **prebuilt binary may break at runtime** because it **can't find the symbol** it needs.

Before Android 10:

- You'd get **runtime crashes**.
- You had to **manually check logs** (very hard and slow).

From Android 10 onwards:
✅ **Build-time check** will detect if prebuilt libraries are broken.

🔎 What does the ABI Usages Checker do?

The **symbol-based ABI usage checker** simulates how the **dynamic linker** resolves symbols inside your `.so` or `.a` files **during build**.

It checks 4 things:

1. **Architecture**
   - Only checks `arm`, `arm64`, `x86`, `x86_64`.
   - Other architectures are skipped.
2. **Dependencies declared?**
   - Your prebuilt must list all its dependencies in:
     - `shared_libs` (Android.bp)
     - `LOCAL_SHARED_LIBRARIES` (Android.mk)

   ✅ Example:

   ```bp
   cc_prebuilt_library_shared {
       name: "libmy_prebuilt",
       srcs: ["libmy_prebuilt.so"],
       shared_libs: ["libc", "liblog", "libm"], // Required!
   }
   ```
3. **DT_NEEDED matches?**
   - The `.so` file internally says which libraries it needs (via `DT_NEEDED` in ELF).
   - The checker matches those with `shared_libs`.
   - If mismatch → ⚠️ **error!**
4. **Are all undefined symbols resolved?**
   - Every external symbol used by the `.so` must be found in one of the listed `shared_libs`.
   - If a symbol like `log_print()` is used, but not present in any linked library → ❌ ABI error.

🔧 How to Fix?

A. Declare correct dependencies

```bp
shared_libs: ["libc", "libdl", "liblog"],
```

Or in Make:

```mk
LOCAL_SHARED_LIBRARIES := libc libdl liblog
```

B. If your prebuilt has unresolved symbols *on purpose* (e.g., lazy loaded at runtime):

```bp
allow_undefined_symbols: true,
```

Or:

```mk
LOCAL_ALLOW_UNDEFINED_SYMBOLS := true
```

Use only if you know what you're doing.

C. If your binary is not really a .so or not for ELF checking (e.g., placeholder, stub):

```bp
check_elf_files: false,
```

Or:

```mk
LOCAL_CHECK_ELF_FILES := false
```

🧪 Run Checker Separately

To test all prebuilt `.so` files without full build:

```bash
m check-elf-files
```

🛠️ Fix ABI Check Errors Automatically

If your `Android.bp` is broken:

```bash
fix_android_bp_prebuilt <path-to-Android.bp>
# OR fix it in-place
fix_android_bp_prebuilt --in-place <path-to-Android.bp>
```

If using `Android.mk`:

```bash
fix_android_mk_prebuilt <path-to-Android.mk>
# OR in-place:
fix_android_mk_prebuilt --in-place <path-to-Android.mk>
```

It updates with the correct **`shared_libs`**, **`allow_undefined_symbols`**, etc.

📦 Real-World Example

Let's say your **`libcamera_driver.so`** is a prebuilt binary and it depends on **`libc`**, **`libutils`**, and **`libhardware`**.

If you forget to declare **`libutils`**, the checker will say:

```cpp
undefined symbol: android::sp<...>
```

✅ Fix in **`Android.bp`**:

```bp
cc_prebuilt_library_shared {
    name: "libcamera_driver",
    srcs: ["libcamera_driver.so"],
    shared_libs: ["libc", "libutils", "libhardware"],
}
```

🧩 Summary Table

Feature	Description
`check-elf-files`	Verifies all ELF prebuilt **`.so`** symbols
`allow_undefined_symbols`	Allows unresolved symbols (rare)
`check_elf_files: false`	Skips checking this prebuilt
`fix_android_bp_prebuilt`	Auto-fixes ABI errors in Android.bp
DT_NEEDED	What the **`.so`** says it depends on internally
shared_libs	What you declare in build file

Would you like me to show how to **test this on a real `.so` file using `readelf` or `nm`** tools? Or how to write a complete **`Android.bp`** for a prebuilt **`.so`** with proper ABI safety?

17. explain in simple words with example

Sure! Let's break down the **Prebuilt ABI Usage Checker** in super simple words with a **realistic example**:

🧠 What's the Problem?

Suppose you are a developer who uses a **prebuilt `.so` library** (like **`libmyfilter.so`**) in your Android project. This library was compiled somewhere else and you're **not building it from source**.

Now this **`.so`** file **depends on other libraries** like **`libc`**, **`liblog`**, or **`libm`**. These libraries live in Android's system partition and may change when Android updates.

So, if one of these system libraries **removes or changes a function**, your prebuilt library can **crash at runtime** with errors like:

```javascript
undefined symbol: log_print
```

👉 That's bad! And hard to catch.

✅ What's the Solution?

Android 10+ introduces a **Prebuilt ABI Usage Checker**, which checks your **`.so`** **at build time** to catch these problems early — **before they crash your app or system**.

🧪 Example (Step-by-Step)

Imagine you have this prebuilt library:

```csharp
libmyfilter.so  ← already compiled (you don't build this from source)
```

You write this **`Android.bp`** to include it:

```bp
cc_prebuilt_library_shared {
    name: "libmyfilter",
    srcs: ["libmyfilter.so"],
    shared_libs: ["libc", "liblog", "libm"],  // Required!
}
```

Inside **`libmyfilter.so`**, it uses this function:

```c
#include <log/log.h>

void my_log() {
    __android_log_print(ANDROID_LOG_INFO, "TAG", "Hello");
}
```

💥 What If You Forget Something?

Let's say you forget to include **`liblog`** in **`shared_libs`**:

```bp
shared_libs: ["libc", "libm"],  // ❌ liblog missing
```

Then the checker will run and say something like:

```javascript
undefined symbol: __android_log_print in libmyfilter.so
```

✅ So now you **know the problem at build time**, not after flashing and running.

🛠️ Fix It

Just add the missing dependency:

```bp
shared_libs: ["libc", "liblog", "libm"],
```

Done! Now the checker is happy, and your prebuilt `.so` won't crash later.

🔍 Extra Configs

If your `.so` intentionally uses dynamic loading (`dlsym`) or doesn't resolve all symbols:

```bp
allow_undefined_symbols: true,
```

Use only if you really know what you're doing.

If your `.so` is not really an ELF binary (or not for Android runtime):

```bp
check_elf_files: false,
```

🧰 Auto Fix Tool

You can fix broken `Android.bp` automatically:

```bash
fix_android_bp_prebuilt path/to/Android.bp
```

Or update in-place:

```bash
fix_android_bp_prebuilt --in-place path/to/Android.bp
```

It adds the missing `shared_libs` for you.

🏁 Summary

✅ You Have	✅ What To Do
Prebuilt `.so` file	Add `cc_prebuilt_library_shared` in Android.bp
It uses system functions	Declare `shared_libs` like `"libc"`, `"liblog"`
Build error shows "undefined symbol"	Add the missing dependency
Want to skip symbol check	Use `check_elf_files: false`
Know some symbols will fail (e.g., using `dlsym`)	Use `allow_undefined_symbols: true`

Would you like me to show a complete working example project with a fake `.so` and build file?