1. Vendor interface object

bookmark_border
This document describes the design of the vendor interface object (VINTF object), which aggregates relevant information about a device and makes that information available through a queryable API.

VINTF object design
A VINTF object gathers some of the information it needs directly from the device. Other aspects, such as the manifests, are described statically in XML.

Figure 1. Manifests, compatibility matrixes, and runtime-collectible information.

VINTF object design provides the following for device and framework components:

For the Device	For the Framework
Defines a schema for the static component (the device manifest file).
Adds build time support for defining the device manifest file for a given device.
Defines the queryable API at runtime that retrieves the device manifest file (along with the other runtime-collectible information) and packages them into the query result.
Defines a schema for the static component (the framework manifest file).
Defines the queryable API at runtime that retrieves the framework manifest file and packages it into the query result.
The VINTF object must be reliable and provide the same complete information regardless of when the object is requested (see Caveats).

Manifests and matrixes
As of Android 8.0, a runtime API queries what is on the device and sends that information to the Over-the-Air (OTA) update server and other interested parties (such as CTS DeviceInfo). Some information is retrieved at runtime and some of it is statically-defined.

The device manifest describes the static component of what the device can provide to the framework.
The framework compatibility matrix describes what the Android framework expects from a given device. The matrix is a static entity whose composition is determined manually during development of the next release of the Android framework.
The framework manifest describes high-level services the framework can provide to the device.
The device compatibility matrix describes the services the vendor image requires of the framework. Its composition is determined manually during the development of the device.
These two pairs of manifests and matrixes must be reconciled at OTA time to ensure a device can get framework updates that are compatible with the device's capabilities. In general, a manifest describes what is provided and a compatibility matrix describes what is required.

This section includes the following details on manifests and matrixes:

Manifests defines the device manifest, framework manifest, and manifest file schema.
Compatibility Matrixes defines the schema for the compatibility matrix.
FCM Lifecycle details how HIDL HALs are deprecated and removed and how FCM files are modifed to reflect the status of the HAL Version.
DM Development describes how vendors can define and declare the Target FCM Version in the device manifest for new devices or implement new HAL versions and increment the Target FCM Version when upgrading the vendor image for old devices.
Matching Rules defines the rules for a successful match between a compatibility matrix and a manifest.

Let's break down the **Vendor Interface Object (VINTF)** concept in simple terms, using examples to help clarify.

🔍 What is VINTF?

VINTF (Vendor Interface Object) is a system introduced in Android 8.0 to **standardize communication between the Android framework (system image)** and the **vendor implementation (vendor image)** — essentially between Google's part of Android and the OEM/vendor's part.

Think of it as a **contract** that both sides must agree on to ensure **Android updates work** smoothly on a device without breaking things.

🧩 What does VINTF do?

It **gathers all compatibility information** from a device and makes it available via a **queryable API**. This information includes:

- What HALs (Hardware Abstraction Layers) are available.
- What versions of those HALs are implemented.

- What the system expects from the hardware.
- Whether these match, and thus whether OTA updates will work.

🔄 The Components of VINTF

VINTF relies on **four XML files** to describe what's required and what's provided:

Source	What it describes	File name (usually)
📱 Device	What the device **provides**	`manifest.xml` (device manifest)
📱 Device	What the device **requires from framework**	`compatibility_matrix.xml` (device matrix)
🧠 Framework	What the framework **provides**	`framework_manifest.xml`
🧠 Framework	What the framework **requires from device**	`framework_compatibility_matrix.xml`

📌 Real-World Analogy:

Imagine the Android **framework** is a **driver**, and the **vendor** is a **car**.

- The **driver** (framework) brings a manual (compatibility matrix) saying:
  - "I can drive cars that have auto gear, ABS, and GPS."
- The **car** (device) comes with a spec sheet (manifest) saying:
  - "I have auto gear and ABS but no GPS."

If the driver's requirements **match** the car's features, then he can drive it (i.e., Android can run).

If not, Android can't update without issues.

🔁 OTA Update Use

During an OTA update, Android checks:

- Is the **framework compatibility matrix** compatible with the **device manifest**?
- Is the **device compatibility matrix** compatible with the **framework manifest**?

If both **match**, the update proceeds. If not, it may **block the update** to avoid incompatibility.

🔍 VINTF API

Android provides a **runtime API** to query:

- What HALs are on the device?
- What services the framework is offering?
- Whether they are compatible.

This is used by:

- **OTA update logic**
- **CTS (Compatibility Test Suite)**
- **Developers** or tools like `lshal` to debug HALs

🔧 Build Time and Runtime Info

Info Type	Source	Example
Static	XML files	HAL versions, interface names
Runtime-collected	System runtime	Kernel version, properties, etc.

📅 FCM Lifecycle

FCM = Framework Compatibility Matrix

- Android tracks **HAL deprecation/removal** using FCM versions.
- Vendors must **declare a Target FCM Version** in the device manifest.
  - New HAL added → Update manifest and increment Target FCM
  - HAL removed → Update compatibility matrix to reflect

✅ Summary Rules

- **Manifest** = "I provide this"
- **Compatibility Matrix** = "I require this"
- Matching happens **both ways**:
  - Device manifest ↔ Framework compatibility matrix
  - Framework manifest ↔ Device compatibility matrix

🔗 Example:

```xml
<!-- From device manifest -->
<hal format="hidl" name="android.hardware.camera" version="3.4" />

<!-- From framework compatibility matrix -->
<hal format="hidl" name="android.hardware.camera" version="3.4" optional="false" />
```

✔️ This is a match. So Android framework and vendor code are compatible for this HAL.

🔚 Final Takeaway

VINTF ensures that vendor implementations and Android framework are compatible, even after OTA updates. It automates and formalizes the check using manifests and matrices, avoiding device brick issues after updates.

Let me know if you want:

- XML schema format
- Real device example (`lshal` or `/vendor/manifest.xml`)
- How to write or test your own VINTF XMLs

2. Manifests

bookmark_border
A VINTF object aggregates data from device manifest and framework manifest files (XML). Both manifests share a format, although not all elements apply to both (for details on the schema, see Manifest file schema).

Device manifest
The device manifest (provided by the device) consists of the vendor manifest and the ODM manifest.

The vendor manifest specifies HALs, SELinux policy versions, etc. common to an SoC. It is recommended to be placed in the Android source tree at device/VENDOR/DEVICE/manifest.xml, but multiple fragment files can be used. For details, see Manifest fragments and Generate DM from fragments.
The ODM manifest lists HALs specific to the product in the ODM partition. The VINTF object loads the ODM manifest in this order:
If SKU is defined (where SKU is the value of the property ro.boot.product.hardware.sku), /odm/etc/vintf/manifest_SKU.xml
/odm/etc/vintf/manifest.xml
If SKU is defined, /odm/etc/manifest_SKU.xml
/odm/etc/manifest.xml
The vendor manifest lists HALs specific to the product in the vendor partition. The VINTF object loads the vendor manifest in this order:
If SKU is defined (where SKU is the value of the property ro.boot.product.vendor.sku), /vendor/etc/vintf/manifest_SKU.xml
/vendor/etc/vintf/manifest.xml
The VINTF object loads the device manifest in this order:
If the vendor manifest exists, combine the following:
The vendor manifest
Optional vendor manifest fragments
Optional ODM manifest
Optional ODM manifest fragments
Otherwise, if the ODM manifest exists, combine the ODM manifest with the optional ODM manifest fragments.
/vendor/manifest.xml (legacy, no fragments)
Finally, combine manifest fragments from any vendor APEXes. Manifest fragments are loaded from the etc/vintf directory of each APEX (e.g. /apex/<apex name>/etc/vintf).
Note that:

On legacy devices, the legacy vendor manifest and the ODM manifest are used. The ODM manifest may completely override the legacy vendor manifest.
On devices launched with Android 9, the ODM manifest is combined with the vendor manifest.
When combining a list of manifests, manifests that appear later in the list may override tags in manifests that appear earlier in the list, provided that the tags in the later manifest have the attribute override="true". For example, the ODM manifest may override some <hal> tags from the vendor manifest. See the documentation for attribute override below.
This setup enables multiple products with the same board to share the same vendor image (which provides common HALs) yet have different ODM images (which specify product-specific HALs).

Here's an example vendor manifest.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Comments, Legal notices, etc. here -->
<manifest version="2.0" type="device" target-level="1">
    <hal>
        <name>android.hardware.camera</name>
        <transport>hwbinder</transport>
        <version>3.4</version>
        <interface>
            <name>ICameraProvider</name>
            <instance>legacy/0</instance>
            <instance>proprietary/0</instance>
        </interface>
    </hal>
    <hal>
        <name>android.hardware.nfc</name>
        <transport>hwbinder</transport>
        <version>1.0</version>
        <version>2.0</version>
        <interface>
```

```
            <name>INfc</name>
            <instance>nfc_nci</instance>
        </interface>
    </hal>
    <hal>
        <name>android.hardware.nfc</name>
        <transport>hwbinder</transport>
        <fqname>@2.0::INfc/default</fqname>
    </hal>
    <hal>
        <name>android.hardware.drm</name>
        <transport>hwbinder</transport>
        <version>1.0</version>
        <interface>
            <name>ICryptoFactory</name>
            <instance>default</instance>
        </interface>
        <interface>
            <name>IDrmFactory</name>
            <instance>default</instance>
        </interface>
        <fqname>@1.1::ICryptoFactory/clearkey</fqname>
        <fqname>@1.1::IDrmFactory/clearkey</fqname>
    </hal>
    <hal format="aidl">
        <name>android.hardware.light</name>
        <version>1</version>
        <fqname>ILights/default</fqname>
    </hal>
    <hal format="aidl">
        <name>android.hardware.power</name>
        <version>2</version>
        <interface>
            <name>IPower</name>
            <instance>default</instance>
        </interface>
    </hal>
    <hal format="native">
        <name>EGL</name>
        <version>1.1</version>
    </hal>
    <hal format="native">
        <name>GLES</name>
        <version>1.1</version>
        <version>2.0</version>
        <version>3.0</version>
    </hal>
    <sepolicy>
        <version>25.0</version>
    </sepolicy>
</manifest>
```

Here's an example ODM manifest.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Comments, Legal notices, etc. here -->
<manifest version="1.0" type="device">
    <!-- camera 3.4 in vendor manifest is ignored -->
    <hal override="true">
        <name>android.hardware.camera</name>
        <transport>hwbinder</transport>
        <version>3.5</version>
        <interface>
            <name>ICameraProvider</name>
            <instance>legacy/0</instance>
        </interface>
```

```
    </hal>
    <!-- NFC is declared to be disabled -->
    <hal override="true">
        <name>android.hardware.nfc</name>
        <transport>hwbinder</transport>
    </hal>
    <hal>
        <name>android.hardware.power</name>
        <transport>hwbinder</transport>
        <version>1.1</version>
        <interface>
            <name>IPower</name>
            <instance>default</instance>
        </interface>
    </hal>
</manifest>
```

Here's an example device manifest in an OTA package.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Comments, Legal notices, etc. here -->
<manifest version="1.0" type="device" target-level="1">
    <!-- hals ommited -->
    <kernel version="4.4.176">
        <config>
            <key>CONFIG_ANDROID</key>
            <value>y</value>
        </config>
        <config>
            <key>CONFIG_ARM64</key>
            <value>y</value>
        </config>
    <!-- other configs ommited -->
    </kernel>
</manifest>
```

For more details, see Device Manifest Development.

Framework manifest
The framework manifest file consists of the system manifest, the product manifest, and the system_ext manifest.

The system manifest (provided by Google) is manually generated and lives in the Android source tree at /system/libhidl/manifest.xml.
The product manifest (provided by the device) lists HALs serviced by modules installed on the product partition.
The system_ext manifest (provided by the device) lists the following:
HALs serviced by modules installed on the system_ext partition;
VNDK versions;
System SDK version.
Similar to the device manifest, multiple fragment files can be used. For details, see Manifest fragments.

Here's an example framework manifest.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Comments, Legal notices, etc. here -->
<manifest version="1.0" type="framework">
    <hal>
        <name>android.hidl.allocator</name>
        <transport>hwbinder</transport>
        <version>1.0</version>
        <interface>
            <name>IAllocator</name>
            <instance>ashmem</instance>
        </interface>
    </hal>
    <hal>
        <name>android.hidl.memory</name>
        <transport arch="32+64">passthrough</transport>
```

```
        <version>1.0</version>
        <interface>
            <name>IMapper</name>
            <instance>ashmem</instance>
        </interface>
    </hal>
    <hal>
        <name>android.hidl.manager</name>
        <transport>hwbinder</transport>
        <version>1.0</version>
        <interface>
            <name>IServiceManager</name>
            <instance>default</instance>
        </interface>
    </hal>
    <hal>
        <name>android.frameworks.sensorservice</name>
        <transport>hwbinder</transport>
        <version>1.0</version>
        <interface>
            <name>ISensorManager</name>
            <instance>default</instance>
        </interface>
    </hal>
    <hal max-level="5">
        <name>android.frameworks.schedulerservice</name>
        <transport>hwbinder</transport>
        <version>1.0</version>
        <interface>
            <name>ISchedulingPolicyService</name>
            <instance>default</instance>
        </interface>
    </hal>
    <vendor-ndk>
        <version>27</version>
    </vendor-ndk>
    <system-sdk>
        <version>27</version>
    </system-sdk>
</manifest>
```

Manifest fragments
In Android 10 and higher, you can associate a manifest entry with a HAL module in the build system. This makes it easier to conditionally include the HAL module in the build system.

Example
In your Android.bp or Android.mk file, add vintf_fragments to any module that is explicitly installed on the device, like cc_binary or rust_binary. For example, you can modify the module with your implementation of your HAL (my.package.foo@1.0-service-bar).

```
... {
    ...
    vintf_fragments: ["manifest_foo.xml"],
    ...
}

LOCAL_MODULE := ...
LOCAL_VINTF_FRAGMENTS := manifest_foo.xml
```

In a file called manifest_foo.xml, create the manifest for this module. At build time, this manifest is added to the device. Adding an entry here is the same as adding an entry in the device's main manifest. This allows clients to use the interface and allows VTS to identify which HAL implementations are on the device. Anything that a regular manifest does, this manifest also does.

The example below implements android.hardware.foo@1.0::IFoo/default, which is installed to the vendor or odm partition. If it's installed to the system, product, or system_ext partition, use type framework instead of type device.

```
<manifest version="1.0" type="device">
```

```
  <hal format="hidl">
    <name>android.hardware.foo</name>
    <transport>hwbinder</transport>
    <fqname>@1.0::IFoo/default</fqname>
  </hal>
</manifest>
```

If a HAL module is packaged in a vendor APEX, package its associated VINTF fragments within the same APEX with prebuilt_etc as explained in VINTF fragments.

Manifest file schema

This section describes the meaning of these XML tags. Some "required" tags can be missing from the source file in Android source tree and written by assemble_vintf at build time. Required tags must be present in the corresponding files on the device.

?xml
Optional. Only provides information to the XML parser.
manifest.version
Required. Meta-version of this manifest. Describes the elements expected in the manifest. Unrelated to XML version.
manifest.type
Required. Type of this manifest. It has the value device for the device manifest file and framework for the framework manifest file.
manifest.target-level
Required for the device manifest. Specifies the framework compatibility matrix (FCM) version that this device manifest is targeted to be compatible with. This is also called the shipping FCM version of the device.
manifest.hal
Optional, can repeat. A single HAL (HIDL or native, such as GL), depending on the format attribute.
manifest.hal.format
Optional. The value can be one of:
hidl: HIDL HALs. This is the default.
aidl: AIDL HALs. Only valid at manifest meta-version 2.0 and above.
native: Native HALs.
manifest.hal.max-level
Optional. Only valid on framework manifests. If set, HALs with a max level lower than the Target FCM Version in the framework manifest are disabled.
manifest.hal.override
Optional. The value can be one of:
true: Override other <hal> elements with the same <name> and major version. If no <version> or <fqname> are in this <hal> element, then the <hal> element declares this HAL to be disabled.
false: Don't override other <hal> elements with the same <name> and major version.
manifest.hal.name
Required. Fully qualified package name of the HAL. Multiple HAL entries can use the same name. Examples:
android.hardware.camera (HIDL or AIDL HAL)
GLES (native HAL, requires name only)
manifest.hal.transport
Required when manifest.hal.format == "hidl". Must NOT be present otherwise. States what transport is used when an interface from this package is queried from the service manager. The value can be one of:
hwbinder: Binderized mode
passthrough: Passthrough mode
Optional when manifest.hal.format == "aidl". Must NOT be present otherwise. States what transport is used when an interface is served remotely. The value must be:
inet: Inet socket
The manifest.hal.transport.ip and manifest.hal.transport.port must be used to further specify the Inet connection information.
manifest.hal.transport.arch
Required for passthrough and must not be present for hwbinder. Describes the bitness of the passthrough service being provided. The value can be one of:
32: 32-bit mode
64: 64-bit mode
32+64: Both
manifest.hal.transport.ip
Required for inet and must NOT be present otherwise. Describes the IP address from which the remote interface is being served.
manifest.hal.transport.port
Required for inet and must NOT be present otherwise. Describes the port from which the remote interface is being served.
manifest.hal.version
Optional, can repeat. A version for the hal tags in a manifest.

For HIDL and native HALs, the format is MAJOR.MINOR. For examples, refer to hardware/interfaces, vendor/${VENDOR}/interfaces, frameworks/hardware/interfaces, or system/hardware/interfaces.

HIDL and native HALs may use multiple version fields as long as they represent distinct major versions, with only one minor version per major version provided. For example, 3.1 and 3.2 can't coexist, but 1.0 and 3.4 can. This applies to all hal elements with the same name, unless override="true". The values of <version> are not associated with <fqname> because a <fqname> carries a version.

For AIDL HALs, <version> must not be present on devices running Android 11 and below. <version> must be a single integer on devices running Android 12 and above. There must at most one <version> for each (package, interface, instance) tuple. If not present, default to 1. The value of <version> is associated with all <fqname> in the same <hal> because a <fqname> does not carry a version.
manifest.hal.interface
Required, can repeat without duplicates. State an interface in the package that has an instance name. There can be multiple <interface> elements in a <hal>; names must be distinct.
manifest.hal.interface.name
Required. Name of the interface.
manifest.hal.interface.instance
Required, can repeat. Instance name of the interface. Can have multiple instances for an interface but no duplicated <instance> elements.
manifest.hal.fqname
Optional, can repeat. An alternative way to specify an instance for the HAL with name manifest.hal.name.
For HIDL HALs, the format is @MAJOR.MINOR::INTERFACE/INSTANCE.
For AIDL HALs, the format is INTERFACE/INSTANCE.
manifest.sepolicy
Required. Contains all sepolicy-related entries.
manifest.sepolicy.version
Required for the device manifest. Declares the SELinux version. It has the format SDK_INT.PLAT_INT.
manifest.vendor-ndk
Required, can repeat; required for the framework manifest. Must not be present in the device manifest. Multiple <vendor-ndk> entries must have different <version>s. Describes a set of VNDK snapshots provided by the framework.
manifest.vendor-ndk.version
Required. This is a positive integer representing the version of the VNDK snapshot.
manifest.vendor-ndk.library
Optional, can repeat, without duplicates. Describes a set of VNDK libraries provided by the framework for this VNDK vendor snapshot. The value is the filename of a library, e.g. libjpeg.so, including the prefix lib and the suffix .so. No path components are allowed.
manifest.system-sdk.version
Optional, can repeat, without duplicates; used only by the framework manifest. Describes a set of system SDK versions provided by the framework to vendor apps.
manifest.kernel
Optional. Describes static information about the kernel.
manifest.kernel.target-level
Optional. Describes the kernel branch. Its value defaults to manifest.target-level if not present. Must be greater than or equal to manifest.target-level. See kernel match rules for details.

Here's a **simplified breakdown** of everything you need to know about **VINTF Manifests** (Device and Framework), how they work, how they are loaded, and how they relate to OTA updates — explained with examples.

🔍 What are VINTF Manifests?

VINTF (Vendor Interface) uses **XML manifest files** to declare:

- What the **device provides** → **Device Manifest**
- What the **Android framework provides** → **Framework Manifest**

Both follow the **same schema format**, with some tags specific to one side.

🧱 Two Types of Manifests

Manifest Type	Description	Lives In	Purpose
Device Manifest	HALs & kernel/sepolicy info **offered by the device**	`/vendor`, `/odm`, vendor APEXes	What vendor code provides
Framework Manifest	HALs & SDK info **offered by the Android system**	`/system`, `/product`, `/system_ext`	What system image provides

---

📦 Device Manifest Components

Device manifest = Combination of:

- **Vendor manifest** (SoC/common HALs)
- **ODM manifest** (product-specific HALs)
- **Manifest fragments** (e.g. from APEXes)

🔄 Loading Order:

```plaintext
1. /vendor/etc/vintf/manifest_SKU.xml
2. /vendor/etc/vintf/manifest.xml
3. /odm/etc/vintf/manifest_SKU.xml
4. /odm/etc/vintf/manifest.xml
5. /vendor/manifest.xml (legacy fallback)
6. APEX fragments: /apex/<apex>/etc/vintf/*.xml
```

Fragments and later manifests can **override** earlier ones using `override="true"`.

✅ Example Use Case

Say your **vendor manifest** defines:

```xml
<hal>
  <name>android.hardware.camera</name>
  <version>3.4</version>
</hal>
```

Then your **ODM manifest** can **override** it like:

```xml
<hal override="true">
  <name>android.hardware.camera</name>
  <version>3.5</version>
</hal>
```

That means:
📱 Final HAL version = **3.5** (from ODM)

🧱 Framework Manifest Components

Framework manifest = Combination of:

- **System manifest**: `/system/libhidl/manifest.xml`
- **Product manifest**: `/product/etc/vintf/manifest.xml`
- **System_ext manifest**: `/system_ext/etc/vintf/manifest.xml`

Includes:

- HALs served by system/product/system_ext
- VNDK version
- System SDK version

🧩 What is a Manifest Fragment?

A **manifest fragment** is a small VINTF snippet tied to a HAL module. It's useful for modularization (especially with APEX modules).

📦 Example

You have a HAL implementation for `android.hardware.foo`. In its `Android.bp`:

```bp
cc_binary {
  name: "android.hardware.foo-service",
  vintf_fragments: ["manifest_foo.xml"],
}
```

And **`manifest_foo.xml`**:

```xml
<manifest version="1.0" type="device">
  <hal format="hidl">
    <name>android.hardware.foo</name>
    <transport>hwbinder</transport>
    <fqname>@1.0::IFoo/default</fqname>
  </hal>
</manifest>
```

This gets **auto-included** in the final device manifest during the build.

📐 Manifest Schema (Important Tags)

Tag	Meaning
`<manifest version="..." type="...">`	**`type=device`** or **`framework`**
`<hal>`	Declares one HAL
`<hal name>`	Package name (e.g. **`android.hardware.camera`**)
`<version>`	HAL version: **`MAJOR.MINOR`** (HIDL), **`int`** (AIDL)
`<transport>`	**`hwbinder`** (Binderized) or **`passthrough`**
`<interface><name>...<instance>...`	Declares available interfaces
`<fqname>`	Shorthand for **`@version::Interface/Instance`**
`<sepolicy><version>`	SELinux policy version: e.g. **`25.0`**
`<vendor-ndk>` / **`<system-sdk>`**	Declares SDK/VNDK version in framework manifest
`<kernel>`	Kernel version and **`CONFIG_*`** entries

🧪 OTA & Compatibility

At OTA time:

- Android checks that device manifest matches framework compatibility matrix.
- If they don't match → **OTA is blocked**.

🧠 Practical Scenarios

✅ Good Match:

- Device manifest offers:

  ```ruby
  android.hardware.light@1.0::ILight/default
  ```

- Framework requires:

  ```ruby
  android.hardware.light@1.0::ILight/default
  ```

OTA proceeds.

❌ Bad Match:

- Device offers `android.hardware.camera@3.2`
- Framework requires `camera@3.5`

OTA fails unless ODM manifest overrides and upgrades the version.

✅ Final Summary

Concept	Device Manifest	Framework Manifest
Who writes	OEM/Vendor	Google + OEM
Partition	`/vendor`, `/odm`	`/system`, `/product`
Declares	HALs, kernel, SELinux	HALs, SDKs, VNDK
Format	XML (HIDL, AIDL, native)	Same
Override allowed?	Yes (`override="true"`)	No

Would you like a table that maps these manifest files to AOSP code directories or a hands-on example with actual paths from AOSP?

3. ↪
1\. /vendor/etc/vintf/manifest_SKU.xml 2. /vendor/etc/vintf/manifest.xml 3. /odm/etc/vintf/manifest_SKU.xml 4. /odm/etc/vintf/manifest.xml 5. /vendor/manifest.xml (legacy fallback) 6. APEX fragments: /apex/<apex>/etc/vintf/*.xml
explain these in simple words

Sure! Here's a **simple explanation** of each of the manifest file paths used by VINTF:

🔢 Load Order of Device Manifests (Simplified)

The system reads these files **in this order** to know what the device supports (HALs, kernel, sepolicy, etc.).

✅ 1. `/vendor/etc/vintf/manifest_SKU.xml`

- **What it is**: A **custom vendor manifest** for a specific device variant (called SKU).
- **SKU** is set by boot property: `ro.boot.product.vendor.sku`
- **Why**: Some devices have multiple models (like “Pro”, “Lite”) with different hardware.

🧠 Think of it like:

> "This is the vendor manifest just for Galaxy S25 Ultra Pro edition."

✅ 2. `/vendor/etc/vintf/manifest.xml`

- **What it is**: The **main vendor manifest** used if there's no SKU-specific one.
- **Location**: Inside the **vendor partition**
- **Why**: Declares common HALs for all variants of this board or SoC.

🧠 Think of it like:

> "This is the vendor manifest for all Galaxy S25 models."

✅ 3. `/odm/etc/vintf/manifest_SKU.xml`

- **What it is**: A **custom ODM manifest** for a specific SKU.
- **SKU** is set via `ro.boot.product.hardware.sku`
- **Location**: Inside the **ODM partition**

🧠 Think of it like:

> "Extra product-specific HALs or overrides for Galaxy S25 Ultra Pro."

✅ 4. `/odm/etc/vintf/manifest.xml`

- **What it is**: The **main ODM manifest** if there's no SKU-specific one.
- **Location**: In the **ODM partition**
- **Why**: Contains HALs specific to the **product/device**, not the SoC.

🧠 Think of it like:

> "Extra HALs just for Galaxy S25 Ultra, no matter the variant."

🧓 5. `/vendor/manifest.xml` (legacy fallback)

- **What it is**: Old-style manifest from devices launched **before Android 9**.
- **Used only if** the new-style VINTF files above don’t exist.

🧠 Think of it like:

> "An old backup manifest from the early days."

🧩 6. `/apex/<apex>/etc/vintf/*.xml` (Manifest fragments)

- **What it is**: Small manifest pieces included in **APEX modules**.
- **Why**: Some HALs are delivered as modular APEX packages and declare their support here.

🧠 Think of it like:

> "Each modular APEX HAL says 'I exist' through a tiny manifest."

📊 Final Analogy:

Imagine Android is building a **complete list** of hardware features your phone supports. It checks:

1. Is this a special variant (SKU)? → use `_SKU.xml`
2. No? Use regular `manifest.xml`
3. Collect both vendor-wide and product-specific features
4. Add extra pieces from APEX modules
5. If it's an old device, fall back to the old file

Let me know if you'd like a visual diagram to represent this!

4. ↳
Framework manifest = Combination of: * **System manifest**: `/system/libhidl/manifest.xml` * **Product manifest**: `/product/etc/vintf/manifest.xml` * **System_ext manifest**: `/system_ext/etc/vintf/manifest.xml`
will all these will be merged?

Yes, all of these **framework manifest files will be merged** into a single effective **framework manifest** at runtime by the **VINTF system**.

✅ Framework manifest merging (in simple words):

When the system boots or runs a VINTF query, it does the following:

1. **Reads all 3 files:**

Partition	Manifest Path	Who owns it
System	`/system/libhidl/manifest.xml`	Google (AOSP)
Product	`/product/etc/vintf/manifest.xml`	Device maker
System_ext	`/system_ext/etc/vintf/manifest.xml`	Device maker

2. **Combines** them into **one unified framework manifest**, just like a **puzzle coming together**.

🧠 Why merge all 3?

Because different HALs may live in different partitions:

Partition	Example HALs or Data
System	Core Google-provided HALs
Product	Device-specific Google apps or features
System_ext	Optional OEM extensions (e.g., Xiaomi's camera HALs)

So the merged manifest = the **complete set of services that the framework provides** to the vendor side.

📦 Example use case:

Let's say:

- `android.hardware.light@1.0` is defined in system
- `android.hardware.power@1.2` is defined in system_ext
- `android.hardware.vendor.sensor@1.0` is defined in product

After merging, the framework manifest will list **all three HALs**.

✅ Final Answer:

> ✅ **Yes**, all 3 framework manifests (`system`, `system_ext`, `product`) are **merged at runtime** into one **effective framework manifest** by VINTF.

Would you like a diagram showing how merging works for both framework and device manifests?

5. Compatibility matrixes

bookmark_border
This section describes the framework and device compatibility matrixes and the compatibility matrix schema. For match rules, see Matching Rules.

Framework compatibility matrix (FCM)
The framework compatibility matrix (FCM) describes the requirements of the framework on the device it runs on. The framework compatibility matrix consists of the system compatibility matrix, the product compatibility matrix, and the system_ext compatibility matrix. The requirements of the FCM must be satisfied by the device manifest (requirements enforced at build time, runtime, and in VTS).

The system_ext FCM and the product FCM are complements of device-specific FCM (installed in the system partition).

The device FCM should reflect the requirements of modules in the system partition.
The system_ext FCM should reflect the requirements by modules in the system_ext partition.
The product FCM should reflect the requirements by modules in the product partition.
All of the FCMs should align with an OEM's modifications to the framework in the system, product, and system_ext partitions. For example, if an app installed in the product partition uses a vendor extension of a HAL interface, the HAL interface requirement should be declared in the product FCM.

Example system compatibility matrix file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Comments, Legal notices, etc. here -->
<compatibility-matrix version="1.0" type="framework" level="3">
  <hal>
    <name>android.hardware.camera</name>
    <version>1.0</version>
    <version>3.1-4</version>
    <interface>
      <name>ICameraProvider</name>
      <instance>default</instance>
      <regex-instance>[a-z_]+/[0-9]+</regex-instance>
    </interface>
  </hal>
  <hal>
    <name>android.hardware.nfc</name>
    <version>1.0</version>
    <interface>
      <name>INfc</name>
      <instance>default</instance>
    </interface>
  </hal>
  <hal optional="true">
    <name>android.hardware.graphics.composer</name>
    <version>2.1</version>
    <interface>
      <name>IComposer</name>
      <instance>default</instance>
    </interface>
  </hal>
  <hal format="aidl" optional="true">
    <name>android.hardware.light</name>
    <version>1-2</version>
    <interface>
      <name>ILights</name>
      <instance>default</instance>
    </interface>
  </hal>
  <hal format="native">
    <name>GL</name>
    <version>1.1</version>
```

```
        <version>3.0</version>
    </hal>
    <hal format="native">
        <name>EGL</name>
        <version>1.1</version>
    </hal>
    <kernel version="3.18.51">
        <!-- common configs -->
    </kernel>
    <kernel version="3.18.51">
        <!-- arm specific configs -->
        <condition>
            <config>
                <key>CONFIG_ARM</key>
                <value type="tristate">y</value>
            </config>
        <condition>
        <config>
            <key>CONFIG_A</key>
            <value type="string"></value>
        </config>
        <config>
            <key>CONFIG_B</key>
            <value type="tristate">y</value>
        </config>
    </kernel>
    <kernel version="4.1.22">
        <!-- common configs -->
        <config>
            <key>CONFIG_A</key>
            <value type="string">foo</value>
        </config>
        <config>
            <key>CONFIG_B2</key>
            <value type="int">1024</value>
        </config>
    </kernel>
    <sepolicy>
        <kernel-sepolicy-version>30</kernel-sepolicy-version>
        <sepolicy-version>25.0</sepolicy-version>
        <sepolicy-version>26.0-3</sepolicy-version>
    </sepolicy>
    <avb>
        <vbmeta-version>2.1</vbmeta-version>
    </avb>
    <xmlfile format="dtd">
        <name>media_profile</name>
        <version>1.0</version>
        <path>/system/etc/media_profile_V1_0.dtd</path>
    </xmlfile>
</compatibility-matrix>
```

For more details, see FCM Lifecycle.

Product compatibility matrix
The product FCM is a framework compatibility matrix file in the product partition. The VINTF object joins the product FCM with FCMs in the system and system_ext partitions at runtime.

Example product FCM file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Comments, Legal notices, etc. here -->
<compatibility-matrix version="1.0" type="framework">
    <hal>
        <name>vendor.foo.camera</name>
        <version>1.0</version>
```

```xml
    <interface>
      <name>IBetterCamera</name>
      <instance>default</instance>
    </interface>
  </hal>
</compatibility-matrix>
```

System_ext compatibility matrix

The system_ext FCM is a framework compatibility matrix file in the system_ext partition. The VINTF object joins the system_ext FCM with FCMs in the system and product partitions at runtime. See product compatibility matrix for an example system_ext FCM file.

Device compatibility matrix (DCM)

The device compatibility matrix describes a set of requirements the device expects from the framework (requirements enforced at launch and OTA time).

Example DCM file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Comments, Legal notices, etc. here -->
<compatibility-matrix version="1.0" type="device">
  <hal>
    <name>android.hidl.manager</name>
    <version>1.0</version>
    <interface>
      <name>IServiceManager</name>
      <instance>default</instance>
    </interface>
  </hal>
  <hal>
    <name>android.hidl.memory</name>
    <version>1.0</version>
    <interface>
      <name>IMemory</name>
      <instance>ashmem</instance>
    </interface>
  </hal>
  <hal>
    <name>android.hidl.allocator</name>
    <version>1.0</version>
    <interface>
      <name>IAllocator</name>
      <instance>ashmem</instance>
    </interface>
  </hal>
  <hal>
    <name>android.framework.sensor</name>
    <version>1.0</version>
    <interface>
      <name>ISensorManager</name>
      <instance>default</instance>
    </interface>
  </hal>
  <vendor-ndk>
    <version>27</version>
  </vendor-ndk>
  <system-sdk>
    <version>27</version>
  </system-sdk>
</compatibility-matrix>
```

Compatibility matrix schema

This section describes the meaning of these XML tags. Some "required" tags can be missing from the source file in Android source tree and written by assemble_vintf at build time. "Required" tags must be present in the corresponding files on the device.

?xml

Optional. It only provides information to the XML parser.

compatibility-matrix.version

Required. Meta-version of this compatibility matrix. Describes the elements expected in the compatibility matrix. Unrelated to XML version.
compatibility-matrix.type
Required. Type of this compatibility matrix:
"device": Device compatibility matrix.
"framework": Framework compatibility matrix.
manifest.level
Required for framework compatibility matrix. In Android 12 and higher, allowed in framework compatibility matrix files in the product and system_ext partitions. Specifies the Framework Compatibility Matrix Version (FCM Version) of this file. Don't declare this in device-specific framework compatibility matrix (i.e. DEVICE_FRAMEWORK_COMPATIBILITY_MATRIX_FILE).
compatibility-matrix.hal
Optional and can repeat. Lists a single HAL (HIDL or native) that is required by owner of the compatibility matrix (framework or device) to be present. HAL entries are distinguished by a <name> element; there can be several HAL entries with the same name (implies "and" condition).
compatibility-matrix.hal.format
Optional. Value can be one of:
"hidl": HIDL HALs. This is the default.
"aidl": AIDL HALs. Only valid on compatibility matrix meta-version 2.0.
"native": native HALs.
compatibility-matrix.hal.optional (Android 15 or lower)
Attribute is optional and defaults to false. States whether this HAL is optional to the owner of the compatibility matrix (framework or device). If a <hal> entry is marked as optional, it means the owner can work with this HAL, if present, but does not require it to be present.
Warning: This attribute is deprecated after Android 15 and no longer has any effect. If any HALs are required to be installed, this requirement should be enforced in tests.
compatibility-matrix.hal.name
Required. Full package name of this HAL. Examples:
android.hardware.camera (HIDL or AIDL HAL)
GLES (native HAL, requires name only)
compatibility-matrix.hal.version
A list of version ranges (see HAL matches) that defines what versions the owner of the compatibility matrix (framework or device) expects.

For HIDL and native HALs, required, can repeat without duplicates. Format is one of the following:
MAJOR.MINOR_MIN-MINOR_MAX
MAJOR.MINOR (equivalent to MAJOR.MINOR-MINOR)

For AIDL HALs, must not be present on devices running Android 11 and below. Optional on devices running later versions. If specified, format is one of the following:
VERSION_MIN-VERSION_MAX
VERSION (equivalent to VERSION-VERSION)
If unspecified, the value defaults to 1.
compatibility-matrix.hal.interface
Optional, can repeat. A list of required interfaces of this HAL.
compatibility-matrix.hal.interface.name
Required. Name of the interface.
compatibility-matrix.hal.interface.instance
Optional, can repeat. A list of required instances of this interface.
compatibility-matrix.hal.interface.regex-instance
Optional, can repeat. A list of required instance name patterns on this interface. Use Extended Regular Expression format.
compatibility-matrix.kernel
Optional, can repeat. Specify a list of kernel configs that the framework requires on each kernel version.
Multiple <kernel> with the same <version> can exist to imply "and" relationship. Each <kernel> is a "fragment" of the requirements that are enabled only when <conditions> are met.
compatibility-matrix.kernel.version
Required. Kernel version. Format is VERSION.MAJOR_REVISION.MINOR_REVISION. Version and major revision must match exactly. Minor revision defines the minimum LTS version of the kernel the framework expects.
compatibility-matrix.kernel.condition
Optional. Must not exist for the first <kernel> of each version. Specifies a list of conditions. When the conditions are met, the requirements stated in this <kernel> fragment are enabled.
compatibility-matrix.kernel.config
Optional, can repeat. Lists CONFIG items that must be matched for this kernel version. Each CONFIG item is a key-value pair; config items are distinguished by key.
compatibility-matrix.kernel.config.key
Required. Key name of the CONFIG item. Starts with CONFIG_.
compatibility-matrix.kernel.config.value
Required. Value of the CONFIG item. Format depends on type:
string. Quotes are omitted.
int. Decimal and hexadecimal (must start with 0x or 0X)values are accepted. Interpreted as an 64-bit integer; overflows result in truncation.
(Parser accepts values from -264 + 1 to 264 - 1, 65th bit is truncated; for details refer to the strtoull man page.)

range. Format is [int]-[int], e.g. 10-20. Hexadecimal values are accepted and must start with 0x or 0X. Two boundaries must be an unsigned 64-bit integer.
tristate. Valid values are y, m and n.
compatibility-matrix.kernel.config.value.type
Required. Type of the value of the CONFIG item, one of:
string
int
range
tristate
compatibility-matrix.sepolicy
Required. Contains all sepolicy-related entries. Used only by the framework compatibility matrix.
compatibility-matrix.sepolicy.sepolicy-version
Required, can repeat. Describes the requirement on sepolicy version. Corresponds to manifest.sepolicy.version. Each instance of an element defines a range of sepolicy versions.
compatibility-matrix.sepolicy.kernel-sepolicy-version
Required. Declares the policydb version the framework works with.
compatibility-matrix.avb.vbmeta-version
Optional; used only by the framework compatibility matrix. Declares the AVB version used to sign system.img. Deprecated in Android 10.
compatibility-matrix.vendor-ndk
Optional; used only by the device compatibility matrix. Declares the requirement of the VNDK vendor snapshot. If missing, no VNDK requirement is made on the system image.
compatibility-matrix.vendor-ndk.version
Required. A positive integer that declares a VNDK version required by the vendor image.
compatibility-matrix.vendor-ndk.library
Optional, can repeat. Declares a set of VNDK libraries required by the vendor image. Same semantics as manifest.vendor-ndk.library.
compatibility-matrix.system-sdk.version
Optional, can repeat; used only by the device compatibility matrix. Declares the requirement by vendor apps on System SDK versions. If missing, no System SDK requirement is made on the system image.

Let's break down **Compatibility Matrixes** in **simple words**, with examples and analogies.

🧩 What is a Compatibility Matrix?

Think of it like a **checklist**.

- The **Framework Compatibility Matrix (FCM)** = Checklist created by **Google** or OEM to say:

  > "These things (HALs, kernel configs, sepolicy) **must exist** on the device for Android to work."

- The **Device Compatibility Matrix (DCM)** = Checklist created by the **device** to say:

  > "My device needs the following things from the Android system to function properly."

📚 Types of Compatibility Matrixes

Matrix Type	Who owns it	Where it lives	What it checks for
🧠 FCM (framework)	Google/OEM	system, system_ext, product partitions	What the **framework requires** from device
📱 DCM (device)	Vendor/OEM	vendor partition	What the **device expects** from framework

---

📦 Framework Compatibility Matrix (FCM)

This matrix is part of the Android **framework** (in `/system`, `/system_ext`, `/product`) and defines:

- Which HALs are required
- Which kernel version and configs must exist
- Which sepolicy version is expected
- Which XML files (like `media_profile.dtd`) are required

🧠 Think of FCM as:

> "Android OS is saying: 'If you're a device, you must support these HALs & kernel configs to run me.'"

🧪 Example FCM

```xml
xml

<hal>
  <name>android.hardware.camera</name>
  <version>3.1-4</version>
  <interface>
    <name>ICameraProvider</name>
    <instance>default</instance>
  </interface>
</hal>
```

🗣️ This says:

> "Hey device, I need a camera HAL between version 3.1 and 4, with a service called ICameraProvider/default."

📱 Device Compatibility Matrix (DCM)

This matrix comes **from the device side**, saying:

> "Android OS must provide these HALs and libraries so my hardware can function."

- Declares framework-level services needed by device
- Enforced during OTA and at launch

🧪 Example DCM:

```xml
xml

<hal>
  <name>android.hidl.manager</name>
  <version>1.0</version>
  <interface>
    <name>IServiceManager</name>
    <instance>default</instance>
  </interface>
</hal>
```

🗣️ This means:

> "My device can only work if Android gives me this `IServiceManager/default` service."

🔗 How Do FCM and DCM Work Together?

💡 At OTA or boot time:

- VINTF compares **device manifest** 🆚 **FCM**
- VINTF compares **framework manifest** 🆚 **DCM**
- ✅ If all required services match, boot continues
- ❌ If mismatch, OTA may fail or boot may halt

📚 Components inside a Matrix

Tag	What it Describes	Simple Explanation
`<hal>`	HAL packages and versions	HALs needed
`<kernel>`	Required Linux kernel version/configs	Kernel features
`<sepolicy>`	Required SELinux policy version	Security model
`<avb>`	Verified Boot metadata (deprecated)	AVB config
`<vendor-ndk>`	Vendor NDK version requirements	Vendor libraries
`<system-sdk>`	Android system SDK versions needed by vendor apps	SDK for vendor apps
`<xmlfile>`	Specific XML config files required	e.g., media config

📦 Where Do These Matrices Live?

Matrix	File Location (example)
System FCM	`/system/etc/vintf/compatibility_matrix.xml`
Product FCM	`/product/etc/vintf/compatibility_matrix.xml`
System_ext FCM	`/system_ext/etc/vintf/compatibility_matrix.xml`
Device DCM	`/vendor/etc/vintf/compatibility_matrix.xml`

✅ Summary

Question	Answer
What is FCM?	Framework's requirements for devices
What is DCM?	Device's requirements from framework
Who compares them?	VINTF (runtime & OTA)
What happens on mismatch?	OTA or boot failure
Can they include kernel/sepolicy info?	Yes

Would you like a diagram to show how FCM, DCM, and manifests fit together?

6. FCM lifecycle

bookmark_border
An Android framework release has multiple Framework Compatibility Matrixes (FCMs), one for each upgradable Target FCM Version, that define what the framework may use and Target FCM version requirements. As part of the FCM lifecycle, Android deprecates and removes HIDL HALs, then modifies FCM files to reflect the status of the HAL version.

To enable framework-only OTAs in their own ecosystems, partners who extend vendor interfaces should also deprecate and remove HIDL HALs using the same methods.

Note: For more details on HIDL HALs, see Compatibility matrixes, Matching rules, HIDL HAL versioning.
Terminology
Framework Compatibility Matrix (FCM)
An XML file that specifies framework requirements on conforming vendor implementations. The compatibility matrix is versioned, and a new version is frozen for each framework release. Each framework release contains multiple FCMs.
Platform FCM Versions (SF)
The set of all FCM versions in a framework release. The framework can work with any vendor implementation that satisfies one of these FCMs.
FCM Version (F)
The highest version among all FCMs in a framework release.
Target FCM Version (V)
The targeted FCM version (from SF), declared explicitly in the device manifest, that a vendor implementation satisfies. A vendor implementation must be generated against a published FCM, although it may declare newer HAL versions in its Device Manifest.
HAL Version
A HAL Version has the format foo@x.y, where foo is the HAL name and x.y is the specific version; e.g. nfc@1.0, keymaster@3.0 (the root prefix, e.g. android.hardware, is omitted throughout this document.)
Device Manifest
XML files that specify which HAL versions the device side of the vendor interface, including the vendor and ODM images, provides. The contents of a device manifest are constrained by the Target FCM version of the device but can list HALs that are strictly newer relative to the FC corresponding to V.
Device HALs
HALs that are listed (provided) in the device manifest and listed in the framework compatibility matrix (FCM).
Device Compatibility Matrix (DCM)
An XML file that specifies vendor requirements on conforming framework implementations. Each device contains one DCM.
Framework Manifest
An XML file that specifies which HAL versions the framework side of the vendor interface, including system, system_ext, and product images, provides. HALs in the framework manifest are dynamically disabled according to the device's Target FCM version.
Framework HALs
HALs that are listed as provided in the framework manifest and listed in the device compatibility matrix (DCM).
FCM lifecycle in the codebase
This document describes the FCM lifecycle in the abstract. To see the supported manifests, refer to hardware/interfaces/compatibility_matrix.<FCM>.xml where FCM can be found in system/libvintf/include/vintf/Level.h.

A device shipping the corresponding Android release version is expected to have an FCM value greater than or equal to the equivalent level. For example, a device shipping with Android 11 would generally have FCM level 5, but implement FCM level 6 or greater, which comes with various additional requirements specified in the compatibility matrixes. The supported levels for Android 15 are:

FCM	Android Version
5	Android 11/R
6	Android 12/S
7	Android 13/T
8	Android 14/U
202404	Android 15/V

The FCM level is equal to or newer than the Vendor API Level.

When Android deprecates an FCM level, it's still supported for existing devices. Devices targeting lower FCM levels are implicitly allowed to use HALs listed in newer FCM levels, as long as they are available in the branch.

Develop in a new FCM version
Android increments the FCM version for each framework release (such as Android 8 and 8.1). During development, the new compatibility_matrix.F.xml is created and the existing compatibility_matrix.f.xml (where f < F) is no longer changed.

To start developing in a new FCM Version F:

Copy the latest compatibility_matrix.<F-1>.xml to compatibility_matrix.F.xml.
Update the level attribute in the file to F.
Add corresponding build rules to install this compatibility matrix to the device.
Introduce a new HAL
During development, when introducing a new HAL (Wi-Fi, NFC, etc.) to Android on the current FCM version F, add the HAL to compatibility_matrix.F.xml.

For example, Android 8.1 introduced cas@1.0. Devices launching with Android 8.1 can implement this HAL, so the following entry was added to compatibility_matrix.F.xml (which used to be named compatibility_matrix.current.xml temporarily during development of that release):

```
<hal format="hidl">
   <name>android.hardware.cas</name>
   <version>1.0</version>
   <interface>
      <name>IMediaCasService</name>
      <instance>default</instance>
   </interface>
</hal>
```

Upgrade a HAL (minor)
AIDL HAL versions count as minor HAL versions. HIDL interface versions have major.minor versions like 1.2.

During development, when an AIDL HAL has a version upgrade from 2 to 3 at current FCM Version F, the new version is added to the HAL entry in compatibility_matrix.F.xml. The version field of a HAL entry accepts ranges like 2-3.

For example, Android FCM F introduced foo@3 as a minor version upgrade of the HAL. The older version, foo@2, is used for devices targeting older FCMs while the newer version, foo@3, can be used for devices targeting Android FCM F. The entry in the older FCMs before version 2 look like:

```
<hal format="aidl">
   <name>foo</name>
   <version>2</version>
   <interface>
      <name>IFoo</name>
      <instance>default</instance>
   </interface>
</hal>
```

This entry was copied to compatibility_matrix.F.xml and modified to support version 3 as follows:

```
<hal format="aidl">
   <name>foo</name>
   <version>2-3</version>
   <interface>
      <name>IFoo</name>
      <instance>default</instance>
   </interface>
</hal>
```

Upgrade a HAL (major)
During development, when a HAL has a major-version upgrade at current FCM Version F, the new major version x.0 is added to compatibility_matrix.F.xml with the following settings:

Only version x.0, if devices that ship with V = F must launch with x.0.
With older major versions in the same <hal> tag, if devices that ship with V = F can launch with an older major version.
For example, FCM version F introduces foo@2.0 as a major-version upgrade of the 1.0 HAL and deprecates the 1.0 HAL. The older version, foo@1.0, is used for devices targeting previous FCM versions. Devices targeting FCM version F must provide the new 2.0 version if they provide the HAL. In this example, the previous FCM versions contain this entry:

```
<hal format="hidl">
   <name>foo</name>
   <version>1.0</version>;
   <interface>
      <name>IFoo</name>
      <instance>default</instance>
```

```
  </interface>
</hal>
```

Copy this entry to compatibility_matrix.F.xml and modify as follows:

```
<hal format="hidl">
  <name>foo</name>
  <version>2.0</version>
  <interface>
    <name>IFoo</name>
    <instance>default</instance>
  </interface>
</hal>
```

Restrictions:

Because the 1.0 HAL is not in compatibility_matrix.F.xml, devices that target FCM version F must not provide the 1.0 HAL (as this HAL is considered deprecated).
Because the 1.0 HAL is present in older FCM Versions, the framework can still work with the 1.0 HAL so it backward compatible with old devices that target the older FCM versions.
New FCM versions
The process of releasing an FCM Version on the system partition is done solely by Google as part of an AOSP release and includes the following steps:

Ensure the compatibility_matrix.F.xml has the attribute level="F".
Ensure all devices build and boot.
Update VTS tests to ensure devices launching with the latest framework (based on Shipping API level) have Target FCM Version V >= F.
Publish file to AOSP.
For example, VTS tests ensure that devices launching with Android 9 have Target FCM Version >= 3.

In addition, the product and system_ext FCMs may also list requirements for each platform FCM versions. Release of FCM versions on the product and system_ext partitions is done by the owner of these images, respectively. FCM version numbers on the product and system_ext partitions must align with those on the system partition. Similar to FCM versions on the system partition, the compatibility matrix at FCM version F in the product and system_ext partitions reflects requirements on a device with target FCM version F.

HAL version deprecation
Deprecating a HAL Version is a developer decision (i.e. for AOSP HALs, Google makes the decision). It could happen when a higher HAL version (whether minor or major) is released.

Deprecate a device HAL
When a given device HAL foo@x.y is deprecated at FCM Version F, it means that any device launching with Target FCM Version V = F or later must not implement foo at version x.y or any version older than x.y. A deprecated HAL version is still supported by the framework for upgrading devices.

When FCM Version F is released, a HAL Version foo@x.y is considered deprecated if the specific HAL Version is not explicitly stated in the latest FCM for Target FCM Version V = F. For devices launching with V = F, one of the following conditions is true:

The framework requires a higher version (major or minor);
The framework doesn't require the HAL anymore.
For example, in Android 9, health@2.0 is introduced as a major version upgrade of 1.0 HAL. health@1.0 is removed from compatibility_matrix.3.xml but is present in compatibility_matrix.legacy.xml, compatibility_matrix.1.xml, and compatibility_matrix.2.xml. Hence, health@1.0 is considered deprecated.

Deprecate a framework HAL
When a given framework HAL foo@x.y is deprecated at FCM Version F, it means that any device launching with Target FCM Version V = F or later must not expect the framework to provide foo at version x.y, or at any version older than x.y. A deprecated HAL version is still provided by the framework for upgrading devices.

When FCM version F is released, a HAL Version foo@x.y is considered deprecated if the framework manifest specifies max-level="F - 1" for foo@x.y. For devices launching with V = F, the framework doesn't provide the HAL foo@x.y. The device compatibility matrix on devices launching with V = F must not list framework HALs with max-level < V.

For example, in Android 12, schedulerservice@1.0 is deprecated. Its max-level attribute is set to 5, the FCM version introduced in Android 11. See Android 12 framework manifest.

Removal of support for target FCM versions
When active devices of a certain Target FCM Version V drop below a certain threshold, the Target FCM Version is removed from the set SF of the

next framework release. This is done by both of the following steps:

Removing compatibility_matrix.V.xml from the build rules (so that it isn't installed on the system image), and deleting any code that implemented or depended on the removed capabilities.

Removing framework HALs with max-level lower than or equal to V from the framework manifest, and deleting any code that implements the removed framework HALs.

Devices with a target FCM Version outside of SF for a given framework release cannot upgrade to that release.

Removal of fully deprecated HALs
When an FCM version is removed, some HAL interfaces, or versions of HAL interfaces, are no longer present in any FCMs. This means that Android no longer supports them at all, even for upgrading devices.

After a HAL is no longer supported, developers remove references to that HAL interface from Android, including in the client code in the framework, the default implementation, and VTS test cases.

If there are no supported HALs inheriting from the HAL being removed, then the HAL definition itself can be removed with a few extra steps.

Note: It's common for HIDL HAL interfaces to inherit from previous versions of the interface. AIDL HAL interfaces are implicitly similar.
Remove the HAL interface definition from the source code. This includes the *.aidl files and the Android.bp aidl_interface module.
If HIDL, remove the HASH from hardware/interfaces/current.txt.
If AIDL, remove the aidl_api directory with the frozen AIDL files.
Remove the interface from hardware/interfaces/compatibility_matrices/exclude/fcm_exclude.cpp.
HAL version status
The following sections describe (in chronological order) the possible states of a HAL Version.

Unreleased
For device HALs, if a HAL Version is not in any of the public and frozen compatibility matrixes, it is considered unreleased and possibly in development. This includes HAL Versions that are only in compatibility_matrix.F.xml. Examples:

During the development of Android 9 the health@2.0 HAL was considered an unreleased HAL and was only present in compatibility_matrix.3.xml.
The teleportation@1.0 HAL is not in any released compatibility matrixes, and is also considered an unreleased HAL.
For framework HALs, if a HAL version is only in the framework manifest of an unrelated development branch, it's considered unreleased.

Released and current
For device HALs, if a HAL Version is in any public and frozen compatibility matrix, it is released. For example, after FCM Version 3 is frozen and published to AOSP, the health@2.0 HAL is considered a released and current HAL Version.

If a HAL Version is in a public and frozen compatibility matrix that has the highest FCM Version the HAL version is current (i.e. not deprecated). For example, existing HAL Versions (such as nfc@1.0 introduced in compatibility_matrix.legacy.xml) that continue to exist in compatibility_matrix.3.xml are also considered as released and current HAL Versions.

For framework HALs, if a HAL version is in the framework manifest of the latest released branch without the max-level attribute or (unusually) a max-level equal to or higher than the FCM version released in this branch, it is considered a released and current HAL version. For example, the displayservice HAL is released and current in Android 12, as specified by the Android 12 framework manifest.

Released but deprecated
For device HALs, a HAL Version is deprecated if and only if all of the following are met:

It is released.
It is not in the public and frozen compatibility matrix that has the highest FCM Version.
It is in a public and frozen compatibility matrix that the framework still supports.
Examples:

The health@1.0 HAL is in compatibility_matrix.legacy.xml, compatibility_matrix.1.xml, and compatibility_matrix.2.xml, but not in compatibility_matrix.3.xml. Hence it is considered deprecated in Android 9.
The power HAL has a minor version upgrade in Android 9, but power@1.0 is still in compatibility_matrix.3.xml.
power@1.0 compatibility_matrix.legacy.xml, compatibility_matrix.1.xml, and compatibility_matrix.2.xml.
compatibility_matrix.3.xml has power@1.0-1.
Hence power@1.0 is current, but NOT deprecated, in Android 9.

For framework HALs, if a HAL version is in the framework manifest of the latest released branch with a max-level attribute lower than the FCM version release in this branch, it is considered a released but deprecated HAL version. For example, the schedulerservice HAL is released but deprecated in Android 12, as specified by the Android 12 framework manifest.

Removed
For device HALs, a HAL Version is removed if and only if the following are true:

It was previously released.
It is not in any public and frozen compatibility matrix that the framework supports.
Compatibility matrixes that are public, frozen, but not supported by the framework are kept in the codebase to define the removed HAL Versions set so that VTS tests can be written to ensure removed HALs are not on new devices.

For framework HALs, a HAL version is removed if and only if the following are met:

It was previously released.
It is not in any framework manifest of the latest released branch.
Legacy FCMs
Target FCM Version legacy is a special value for all non-Treble devices. The legacy FCM, compatibility_matrix.legacy.xml, lists the requirements of the framework on legacy devices (i.e. devices launched prior to Android 8.0).

If this file exists for an FCM with version F, any non-Treble device can be upgraded to F provided its device manifest is compatible with this file. Its removal follows the same procedure as FCMs for other Target FCM Versions (removed after the number of active pre-8.0 devices drops below a certain threshold).

Released FCM versions
The list of released FCM versions can be found under hardware/interfaces/compatibility_matrices.

To find the FCM version released with a specific Android release, see Level.h.

Was this helpful?

Here's a **simple explanation** of the **FCM lifecycle** (Framework Compatibility Matrix lifecycle) in Android — like a story about **what Android expects from a device over time**, and how it manages changes.

✅ What is FCM (Recap)?

FCM = **Framework Compatibility Matrix**

Think of it like a **contract** between the **Android OS (framework)** and the **device (vendor HALs)**. It lists all the **HALs**, **kernel settings**, and **SELinux policies** the framework expects from a device.

🧬 Why Do We Need an FCM Lifecycle?

Android evolves over time:

- New HALs are introduced 📦
- Old HALs are deprecated or removed 🗑️
- Kernel/Security requirements change 🔐

So, FCM must also evolve in **versions** (like `compatibility_matrix.5.xml`, `6.xml`, etc.).

📅 Lifecycle Phases of an FCM

Phase	What Happens
Create New FCM	Google copies the previous version → updates HALs, kernel configs, etc.
Introduce New HALs	Add new HALs to the current FCM (`version x.0`, `2-3`, etc.)
Deprecate Old HALs	Remove old HALs from the FCM → marks them as deprecated for future devices
Remove HALs Fully	HAL is deleted from codebase + manifest + tests
Release to AOSP	Google releases FCM `.xml` in AOSP and VTS tests use it to verify devices

🔄 Example Timeline

Imagine a HAL called `foo@1.0`:

Android Version	FCM Level	What Happens with `foo` HAL
Android 8	FCM 2	`foo@1.0` is introduced
Android 9	FCM 3	`foo@2.0` added, `foo@1.0` removed → **deprecated**
Android 10	FCM 4	`foo@1.0` not in any FCM anymore → **fully removed**

🔢 Key Terms in Simple Words

Term	Simple Meaning
Platform FCM Versions (SF)	All FCMs supported by the current Android version
Target FCM Version (V)	What FCM version the device declares it supports
HAL Version	Like `nfc@1.0` or `health@2.0`
Device Manifest	Device's list of provided HALs
Framework Manifest	Android system's provided HALs
Deprecated HAL	HAL is no longer required for new devices but still works on old ones
Removed HAL	HAL is gone completely – no longer allowed

🧪 What Happens During OTA?

- During an **update**, Android checks:
  - "Does the vendor support my Target FCM Version?"
  - "Does the device provide all required HALs in my FCM version?"

If yes ✅: boot/update continues
If no ❌: OTA is blocked

📂 Files Involved

File	Role
`compatibility_matrix.5.xml`	FCM for Android 11
`compatibility_matrix.6.xml`	FCM for Android 12
`device manifest.xml` (vendor side)	Lists what HALs the device provides
`framework manifest.xml` (system side)	Lists what HALs Android provides

🧼 What Does "Deprecate HAL" Mean?

If Android releases a new HAL version (e.g. `health@2.0`), it might:

- Remove `health@1.0` from FCM 3
- Devices launching with FCM 3 or later **must not use** `health@1.0`
- But older devices (e.g. using FCM 1 or 2) can still use it

✅ **Backward compatible**, but not forward compatible

📉 Removal of FCMs

When older FCMs (like FCM 2 or 3) are **no longer used by devices**, Android will:

1. Delete those FCM `.xml` files from AOSP
2. Remove HALs that only exist in those FCMs
3. Disallow OTAs to devices targeting removed FCMs

📘 Summary Table

Term	Meaning
FCM	Android's expectations from the device
DCM	Device's expectations from the Android framework
Target FCM Version (V)	FCM version device says it supports
Released HAL	HAL in public FCM
Deprecated HAL	HAL not in latest FCM, but still in older ones
Removed HAL	HAL no longer in any supported FCM
FCM Version (F)	Latest FCM for the Android version
SF (Platform FCM Set)	All supported FCMs for that Android version

---

Would you like a **diagram** or **flowchart** to show the FCM lifecycle visually?

7. explain in simple words with example

Sure! Let's explain the **FCM lifecycle** in very simple words, step by step, **with a real-world analogy and examples**.

🧠 Think of Android like a Company

- Android (Google) = **Company HQ**
- Phone makers (Samsung, Xiaomi, etc.) = **Branches**
- HALs (hardware interfaces) = **Employees** doing specific jobs (like camera, NFC, Wi-Fi)
- FCM = **Job Requirement Sheet** given by HQ to all branches.

🧾 What is FCM (Framework Compatibility Matrix)?

It's a **list of HALs** (features like camera, Bluetooth, etc.) and **rules** that the Android system expects a phone to support.

For example:

```xml
<hal>
  <name>android.hardware.camera</name>
  <version>1.0</version>
</hal>
```

Means: "Hey phone maker, your device **must support Camera HAL version 1.0** if you want to run Android."

📅 FCM Lifecycle – In Simple Steps

🔧 Step 1: Android Releases a New Version (Say Android 12)

Google creates a new FCM file — let's call it **`compatibility_matrix.6.xml`**.

This file says:

- "Camera HAL must be version 2.0"
- "Bluetooth HAL must be 1.1"
- "Drop support for older camera@1.0"

👉 So, **new phones launching with Android 12 must follow these new rules**.

📘 Example:

Android 10 (FCM version 4)

```xml
<hal>
  <name>android.hardware.camera</name>
  <version>1.0</version>
</hal>
```

Android 11 (FCM version 5)

```xml
<hal>
  <name>android.hardware.camera</name>
  <version>1.0-2.0</version>
</hal>
```

Android 12 (FCM version 6)

```xml
<hal>
  <name>android.hardware.camera</name>
  <version>2.0</version>
</hal>
```

🔄 What This Means:

- Devices launching with Android 10 can use camera@1.0 ✅
- Devices launching with Android 11 can use 1.0 or 2.0 ✅
- Devices launching with Android 12 **must use** 2.0 ❌ for 1.0

📦 Step 2: Deprecation

If Android wants to **stop using an old HAL**, it **removes** it from the FCM file.

So camera@1.0 is **deprecated** in Android 12:

- Not listed in `compatibility_matrix.6.xml`
- Phones launching with FCM 6 **must not use it**

But old phones (Android 10/11) that already have it can still work with Android 12 OTA. ✅

❌ Step 3: Removal

When Android sees **no phone is using camera@1.0**, it removes:

- Code from AOSP
- Tests
- Entries from FCM
- AIDL/HIDL files

Now camera@1.0 is **fully removed** 🚫. No phone, old or new, can use it anymore.

🎯 Final Example Table

FCM Level	Android Version	Camera HAL Required
4	Android 10	camera@1.0 ✅
5	Android 11	camera@1.0 or 2.0 ✅
6	Android 12	camera@2.0 only ✅
7+	Android 13+	camera@1.0 = ❌

👵 What About Legacy Devices?

Old phones that were never Treble-compatible use `compatibility_matrix.legacy.xml`.
These are like:

- Android 7 or older phones
- Still allowed to run, but can't launch new Android versions

---

🔄 Summary in Super Simple Terms

Step	Meaning
Released HAL	Official, allowed by Android for new devices
Deprecated HAL	Still works on old phones, but **not allowed** for new launches
Removed HAL	Deleted from Android; not allowed anywhere
FCM Version	A rulebook for a specific Android version (e.g., FCM 6 = Android 12)

Step	Meaning
Target FCM	What the phone maker says their device follows
Upgrade Check	Android checks FCM match before OTA update

Would you like a **visual diagram** of this FCM lifecycle with HAL versions and Android versions shown step-by-step?

8. Device manifest development

bookmark_border
When developing and releasing new devices, vendors can define and declare the target FCM version in the device manifest (DM). When upgrading the vendor image for old devices, vendors can choose to implement new HAL versions and increment the target FCM version.

Note: For details on terms used in this page, see Terminology.
Develop new devices
When defining the device target FCM version for new devices:

Leave DEVICE_MANIFEST_FILE and PRODUCT_ENFORCE_VINTF_MANIFEST undefined.
Implement HALs for the target FCM version.
Write the correct device manifest file.
Write the target FCM version to device manifest file.
Set DEVICE_MANIFEST_FILE.
Set PRODUCT_ENFORCE_VINTF_MANIFEST to true.
Release new devices
When a new device is released, its initial target FCM version needs to be determined and declared in the device manifest as the "target-level" attribute in the top-level <manifest> element.

For example, devices launching with Android 9 must have target FCM version equal to 3 (the higher version available at this time). To declare this in the device manifest:

```
<manifest version="1.0" type="device" target-level="3">
    <!-- ... -->
</manifest>
```

Upgrade vendor image
When upgrading the vendor image for an old device, vendors can choose to implement new HAL versions and increment the target FCM version.

Upgrade HALs
During a vendor image upgrade, vendors can implement new HAL versions provided that HAL name, interface name, and instance name are the same. For example:

Google Pixel 2 and Pixel 2 XL devices released with target FCM version 2, which implemented the required audio 2.0 HAL android.hardware.audio@2.0::IDeviceFactory/default.
For the audio 4.0 HAL that released with Android 9, Google Pixel 2 and Pixel 2 XL devices can use a full OTA to upgrade to the 4.0 HAL, which implements android.hardware.audio@4.0::IDeviceFactory/default.
Even though the compatibility_matrix.2.xml specifies audio 2.0 only, the requirement on a vendor image with target FCM version 2 has been loosened because the Android 9 framework (FCM version 3) considers audio 4.0 a replacement of audio 2.0 HAL in terms of functionality.
To summarize, given that compatibility_matrix.2.xml requires audio 2.0 and compatibility_matrix.3.xml requires audio 4.0, the requirements are as follows:

FCM version (system)	Target FCM version (vendor)	Requirements
2 (8.1)	2 (8.1)	Audio 2.0
3 (9)	2 (8.1)	Audio 2.0 or 4.0
3 (9)	3 (9)	Audio 4.0

Upgrade target FCM version
During a vendor image upgrade, vendors can also increment the target FCM version to specify the targeted FCM version the upgraded vendor image can work with. To bump the target FCM version of a device, vendors need to:

Implement all new required HAL Versions for the target FCM version.
Modify HAL Versions in the device manifest file.
Modify the target FCM version in the device manifest file.
Remove deprecated HAL versions.
For example, Google Pixel and Pixel XL devices launched with Android 7.0 so their target FCM version must be at least legacy. However, the device manifest declares the target FCM version 2 because the vendor image has been updated to conform with compatibility_matrix.2.xml:

```
<manifest version="1.0" type="device" target-level="2">
```

If vendors do not implement all required new HAL versions or do not remove deprecated HAL versions, the target FCM version cannot be upgraded.

For example, Google Pixel 2 and Pixel 2 XL devices have target FCM version 2. While they do implement some HALs required by compatibility_matrix.3.xml (such as audio 4.0, health 2.0, etc.), they do not remove android.hardware.radio.deprecated@1.0, which is deprecated at FCM version 3 (Android 9). Hence, these devices cannot upgrade the target FCM version to 3.

Mandating kernel requirements during OTA
Updating devices from Android 9 or lower
On devices with Android 9 or lower, ensure the following CLs are cherry-picked:

CL 722283
CL 722284
CL 722345
These changes introduce the build flag PRODUCT_OTA_ENFORCE_VINTF_KERNEL_REQUIREMENTS and leave the flag unset for devices launched with Android 9 or lower.

When updating to Android 10, OTA clients on devices running Android 9 or lower don't check kernel requirements in the OTA package correctly. These changes are needed to drop kernel requirements from the generated OTA package.
When updating to Android 11, it is optional to set the PRODUCT_OTA_ENFORCE_VINTF_KERNEL_REQUIREMENTS build flag to check VINTF compatibility when the update package is generated.
For more information about this build flag, see Updating devices from Android 10.

Updating devices from Android 10
Android 10 introduces a new build flag, PRODUCT_OTA_ENFORCE_VINTF_KERNEL_REQUIREMENTS. For devices launched with Android 10, this flag is automatically set to true. When the flag is set to true, a script extracts the kernel version and the kernel configurations from the installed kernel image.

When updating to Android 10, OTA update package contains kernel version and configuration. OTA clients on devices running Android 10 read this information to check compatibility.
When updating to Android 11, OTA package genreation reads kernel version and configuration to check compatibility.
If the script fails to extract this information for your kernel image, do one of the following:

Edit the script to support your kernel format and contribute to AOSP.
Set BOARD_KERNEL_VERSION to the kernel version and BOARD_KERNEL_CONFIG_FILE to the path of the built kernel configuration file .config. Both variables must be updated when the kernel image is updated.
Alternatively, set PRODUCT_OTA_ENFORCE_VINTF_KERNEL_REQUIREMENTS to false to skip checking kernel requirements. This is not recommended because any incompatibility is hidden and is only discovered when running VTS tests after the update.
You can view the source code of the kernel information extraction script extract_kernel.py.

Here's a **simple explanation** of the **Device Manifest Development** process with **real-world analogies and examples** to help you understand easily:

🧾 What Is a Device Manifest?

The **device manifest** is a file (XML format) where a phone maker tells Android:

- “These are the HALs (hardware support libraries) my device provides.”
- “I’m compatible with a particular Android version (called a target FCM version).”

Think of it like a **resume** for your phone: it lists all its hardware qualifications.

👶 Developing a New Device (From Scratch)

🔲 Steps:

1. **Don’t set** any manifest file initially.
2. Write and implement HALs (camera, audio, etc.) as required for the Android version you're targeting.
3. Create a **`manifest.xml`** like this:

```xml
<manifest version="1.0" type="device" target-level="3">
    <!-- HALs listed here -->
</manifest>
```

4. Set:
   - **`DEVICE_MANIFEST_FILE`** → your manifest path
   - **`PRODUCT_ENFORCE_VINTF_MANIFEST := true`** → to strictly check HAL compatibility

✅ Done! You've now told Android what your phone supports.

🚀 Releasing a New Device

When launching, you must set the **`target-level`** to the **latest FCM level**.

For example:

- Android 9 = FCM level **3**
- Android 10 = level **4**
- Android 11 = level **5**
- Android 15 = level **202404**

Your manifest should look like:

```xml
<manifest version="1.0" type="device" target-level="3">
```

This says: "I follow all the rules of Android 9 (FCM 3)."

🔁 Upgrading an Old Device (Vendor Image)

You have two choices:

1. **Stay on old FCM version**
2. **Move to new FCM version**

✅ Case 1: Stay on Same FCM Version

Suppose your device launched with FCM level 2 (Android 8.1).

You can still update to newer Android (like Android 9), **without changing target FCM, as long as your old HALs are still compatible**.

💡 *Android 9 is flexible. It says: "I can work with old or new HALs."*

System FCM	Device FCM	Required HAL version
3 (Android 9)	2 (Android 8.1)	Audio 2.0 or Audio 4.0 ✅

🚧 Case 2: Upgrade to New FCM Version

To bump your FCM version to 3 (i.e., say you're now a full Android 9 device):

1. **Add new HALs** required by FCM 3 (e.g., **`audio@4.0`**)
2. **Remove deprecated HALs** (e.g., **`radio.deprecated@1.0`**)
3. Update manifest:

```xml
<manifest version="1.0" type="device" target-level="3">
```

🚫 If you keep any old/deprecated HALs → ❌ you **cannot claim** FCM version 3

📦 Real Example

- **Pixel 2** shipped with FCM version 2

- It got OTA update with **`audio@4.0`** (from FCM 3) ✅
- But it still kept **`radio.deprecated@1.0`** ❌
- So it could **not upgrade target FCM** to 3

🧪 Kernel Requirement During OTA

Before Android 10:

- OTA didn't properly check kernel requirements.
- Needed to **skip kernel checks** using build flag:

```makefile
PRODUCT_OTA_ENFORCE_VINTF_KERNEL_REQUIREMENTS := false
```

Android 10+:

- Android introduced strict kernel version/config checks.
- The OTA package includes:
  - Kernel version
  - **`.config`** file values

🔧 If check fails:

- Set **`BOARD_KERNEL_VERSION`** and **`BOARD_KERNEL_CONFIG_FILE`**
- Or disable check (not recommended)

✅ Summary Table

Task	What to Do
New Device	Write manifest, set target-level = current FCM
OTA with old FCM	Keep old HALs, update allowed HALs
Upgrade FCM version	Add new HALs, remove deprecated ones
Enforce manifest	Set **`PRODUCT_ENFORCE_VINTF_MANIFEST := true`**
Kernel check in OTA	Use **`PRODUCT_OTA_ENFORCE_VINTF_KERNEL_REQUIREMENTS`**

🔍 Visual Example: `manifest.xml`

```xml
<manifest version="1.0" type="device" target-level="3">
  <hal>
    <name>android.hardware.audio</name>
    <version>4.0</version>
    <interface>
      <name>IDeviceFactory</name>
      <instance>default</instance>
    </interface>
  </hal>

  <!-- Deprecated HAL should be removed -->
  <!-- <hal name="android.hardware.radio.deprecated" version="1.0" /> -->
</manifest>
```

Let me know if you want:

- a visual diagram of this lifecycle
- or want help **creating a manifest** for your own custom board.

9. ↪
📏 Kernel Requirement During OTA ------------------------------ ### Before Android 10: * OTA didn’t properly check kernel requirements. * Needed to **skip kernel checks** using build flag: makefile CopyEdit `PRODUCT_OTA_ENFORCE_VINTF_KERNEL_REQUIREMENTS := false` ### Android 10+: * Android introduced strict kernel version/config checks. * The OTA package includes: * Kernel version * `.config` file values 🔧 If check fails: * Set `BOARD_KERNEL_VERSION` and `BOARD_KERNEL_CONFIG_FILE` * Or disable check (not recommended)
confusing explain with example

Let’s break down **“📏 Kernel Requirement During OTA”** with a **very simple real-world example** so it's crystal clear.

🤔 Problem

You are a phone manufacturer (say, for “Pixel 2”) and you're pushing an **OTA update** to upgrade Android OS on users' phones. But Android also needs to ensure your **Linux kernel** matches what it expects (right version, correct settings).

Older Android versions **did not check** kernel details properly during OTA. But newer Android versions **enforce strict checking**.

🔧 Real-World Example

Let’s say:

- You made a device in **Android 9 (Pie)** — FCM level 3
- It uses **Linux kernel version 4.9**
- Now, you're pushing an OTA update to **Android 10 (Q)**

Android 9 Behavior:

- It does **not check** your kernel version or config during OTA
- So you can push OTA even if your kernel is outdated or misconfigured ✅

Android 10 Behavior:

- It says: "I will only accept OTA **if your kernel matches** what I expect"
- That means:
  - Your OTA must include:
    - Kernel version info
    - `.config` file (kernel build options like file systems, drivers)
  - OTA updater will **extract and verify** this info

📦 How Kernel Info is Extracted?

Android uses a script:

```bash
extract_kernel.py
```

It reads:

- kernel version
- kernel `.config` values

😫 What if this check fails?

Let’s say:

- Your kernel image format is custom

- Script **can't read** version or **.config** properly
- OTA fails ❌

You now have **3 options**:

✅ Option 1 (Preferred):

You **tell Android manually** about the kernel:

In your device's build config (e.g. **BoardConfig.mk**):

```make
BOARD_KERNEL_VERSION := 4.9
BOARD_KERNEL_CONFIG_FILE := path/to/kernel/.config
```

Now Android uses these values instead of parsing the kernel image.

🧪 Option 2 (Not Recommended):

Disable the check completely:

```make
PRODUCT_OTA_ENFORCE_VINTF_KERNEL_REQUIREMENTS := false
```

⚠️ This skips kernel version/config checks in OTA, **but it's risky**:

- You may push a broken kernel
- Incompatibilities show up only **after** update, when VTS or CTS fail

---

🛠️ Option 3 (Advanced):

Fix the **extract_kernel.py** script to support your custom kernel image.
Then **contribute it back to AOSP**.

🚀 Final Recap:

Android Version	OTA Kernel Check	What to Do
Android 9 or below	❌ No kernel check	Nothing required
Android 10+	✅ Kernel version and config are checked	Add version/config to build or fix script
Can't provide info?	You can disable the check using **PRODUCT_OTA_ENFORCE_VINTF_KERNEL_REQUIREMENTS := false**	

💡 Example in BoardConfig.mk:

```make
BOARD_KERNEL_VERSION := 5.10
BOARD_KERNEL_CONFIG_FILE := device/vendor/mydevice/kernel/.config
```

This tells Android:
"My kernel version is 5.10, and here is the config file. Please use these values during OTA checks."

Would you like me to show you a sample OTA build config for a specific device?

10. ↪
`BOARD_KERNEL_CONFIG_FILE := device/vendor/mydevice/kernel/.config`
what this config will contains

Great question.

🧾 `BOARD_KERNEL_CONFIG_FILE` — What does it contain?

This points to your **kernel build configuration file**, commonly named `.config`.
It is auto-generated during kernel compilation and tells Android what kernel **features, drivers, and options** are enabled.

🔍 Where does `.config` come from?

When you build the kernel with `make` (like `make ARCH=arm64 my_defconfig`), it creates a `.config` file based on your chosen defconfig (like `my_defconfig`).

✅ Contents of `.config`

Here's what it looks like — a **text file with lines like this**:

```make
CONFIG_ARM=y
CONFIG_SMP=y
CONFIG_PREEMPT=y
CONFIG_ANDROID_BINDER_IPC=y
CONFIG_USB_SUPPORT=y
CONFIG_EXT4_FS=m
CONFIG_NETFILTER=y
CONFIG_BT=m
CONFIG_NFC=n
```

Each line means:

- `CONFIG_X=y` → enabled and **built into the kernel**
- `CONFIG_X=m` → enabled as a **loadable module**
- `CONFIG_X=n` → **disabled**

📘 Example Meaning

Line	Meaning
`CONFIG_ARM=y`	Targeting ARM CPUs
`CONFIG_ANDROID_BINDER_IPC=y`	Binder IPC support is included
`CONFIG_EXT4_FS=m`	EXT4 filesystem is a module (loaded on demand)
`CONFIG_USB_SUPPORT=y`	USB support is enabled
`CONFIG_NFC=n`	NFC support is disabled

📦 Why Android needs `.config` in OTA?

To **enforce VINTF kernel requirements**.

During OTA, Android will:

1. Extract your kernel version
2. Read this `.config` file
3. Match against required values in `compatibility_matrix.xml`

Example requirement in FCM:

```xml
<kernel version="4.14.117">
  <config>
    <key>CONFIG_ANDROID_BINDER_IPC</key>
    <value type="tristate">y</value>
  </config>
</kernel>
```

If **`.config`** is missing or has **`CONFIG_ANDROID_BINDER_IPC=n`**, OTA will fail ✅

🛠️ Where to find `.config`?

When you compile the kernel:

```bash
cd kernel/
make ARCH=arm64 vendor_defconfig
make ARCH=arm64
```

After this, you'll get:

```bash
kernel/.config
```

That's the file you point to in:

```make
BOARD_KERNEL_CONFIG_FILE := device/vendor/mydevice/kernel/.config
```

Would you like a full example of **`.config`** content that passes a typical Android OTA check?

11. Match rules

bookmark_border
The two pairs of compatibility matrixes and manifests are meant to be reconciled to verify that the framework and vendor implementation can work with each other. This verification is successful upon a match between the framework compatibility matrix and the device manifest, as well as between the framework manifest and the device compatibility matrix.

This verification is done at build time, at OTA update package generation time, at boot time, and in VTS compatibility tests.

The following sections detail matching rules used by various components.

Framework compatibility matrix version matches
To match a device manifest with a framework compatibility matrix, the shipping FCM version specified by manifest.target-level must be exactly equal to the FCM version specified by compatibility-matrix.level. Otherwise there's no match.

When the framework compatibility matrix is requested with libvintf, this match is always successful because libvintf opens the device manifest, retrieves the shipping FCM version, and returns the framework compatibility matrix at that shipping FCM version (plus some optional HALs from compatibility matrixes at higher FCM versions).

HAL matches
The HAL-match rule identifies the versions of hal elements in a manifest file that are considered to be supported by the owner of the corresponding compatibility matrix.

HIDL and native HALs
The match rules for HIDL and native HALs are as follows:

Multiple <hal> elements are evaluated with a single AND relationship.
<hal> elements can have <hal optional="true"> to mark them as not required.
Warning: The optional tag isn't supported in Android 16 and higher.
Multiple <version> elements within the same <hal> element have the OR relationship. If two or more are specified, only one of the versions needs to be implemented. (See Successful HAL match for DRM module.)
Multiple <instance> and <regex-instance> elements within the same <hal> element are evaluated with a single AND relationship when the <hal> is required. (See Successful HAL match for DRM module.)
Example: Successful HAL match for a module
For a HAL at version 2.5, the match rule is as follows:

Matrix Matching manifest
2.5 2.5-2.∞. In the compatibility matrix, 2.5 is the shorthand for 2.5-5.
2.5-7 2.5-2.∞. Indicates the following:
2.5 is the minimum required version, meaning a manifest providing HAL 2.0-2.4 isn't compatible.
2.7 is the maximum version that can be requested, meaning the owner of the compatibility matrix (framework or device) can't request versions beyond 2.7. The owner of the matching manifest can still serve version 2.10 (as an example) when 2.7 is requested. The compatibility-matrix owner knows only that the requested service is compatible with API version 2.7.
-7 is informational only and doesn't affect the OTA update process.
Thus, a device with a HAL at version 2.10 in its manifest file remains compatible with a framework that states 2.5-7 in its compatibility matrix.
Example: Successful HAL match for DRM module
The framework compatibility matrix states the following version information for DRM HAL:

```
<hal>
  <name>android.hardware.drm
  <version>1.0</version>
  <version>3.1-2</version>
  <interface>
    <name>IDrmFactory</name>
    <instance>default</instance>
    <instance>specific</instance>
  </interface>
</hal>
<hal>
  <name>android.hardware.drm
  <version>2.0</version>
  <interface>
```

```
        <name>ICryptoFactory</name>
        <instance>default</instance>
        <regex-instance>[a-z]+/[0-9]+</regex-instance>
    </interface>
</hal>
```

A vendor must implement ONE of the following instances; either:

```
android.hardware.drm@1.x::IDrmFactory/default          // where x >= 0
android.hardware.drm@1.x::IDrmFactory/specific         // where x >= 0
```

OR:

```
android.hardware.drm@3.y::IDrmFactory/default          // where y >= 1
android.hardware.drm@3.y::IDrmFactory/specific         // where y >= 1
```

AND must implement all of these instances:

```
android.hardware.drm@2.z::ICryptoFactory/default       // where z >= 0
android.hardware.drm@2.z::ICryptoFactory/${INSTANCE}
            // where z >= 0 and ${INSTANCE} matches [a-z]+/[0-9]+
            // e.g. legacy/0
```

AIDL HALs

Android and higher supports versions for AIDL HALs in VINTF. The match rules for AIDL HALs are similar to those of HIDL and native HALs, except that there are no major versions, and there is exactly one version per HAL instance (1 if the version is unspecified):

Multiple <hal> elements are evaluated with a single AND relationship.
<hal> elements can have <hal optional="true"> to mark them as not required.
Warning: The optional tag isn't supported in Android 16 and higher.
Multiple <instance> and <regex-instance> elements within the same <hal> are evaluated with a single AND relationship when the <hal> is required. (See Successful HAL match for multiple modules.)
Example: Successful HAL match for a module
For a HAL at version 5, the match rule is as follows:

Matrix	Matching manifest
5	5-∞. In the compatibility matrix, 5 is the shorthand for 5-5.
5-7	5-∞. Indicates the following:

5 is the minimum required version, meaning a manifest providing HAL 1-4 isn't compatible.
7 is the maximum version that could be requested, meaning the owner of the compatibility matrix (framework or device) won't request versions beyond 7. The owner of the matching manifest can still serve version 10 (as an example) when 7 is requested. The compatibility-matrix owner knows only that the requested service is compatible with API version 7.
-7 is informational only and doesn't affect the OTA update process.
Thus, a device with a HAL at version 10 in its manifest file remains compatible with a framework that states 5-7 in its compatibility matrix.
Example: Successful HAL match for multiple modules
The framework compatibility matrix states the following version information for vibrator and camera HALs:

```
<hal>
    <name>android.hardware.vibrator
    <version>1-2</version>
    <interface>
        <name>IVibrator</name>
        <instance>default</instance>
        <instance>specific</instance>
    </interface>
</hal>
<hal>
    <name>android.hardware.camera
    <version>5</version>
    <interface>
        <name>ICamera</name>
        <instance>default</instance>
        <regex-instance>[a-z]+/[0-9]+</regex-instance>
    </interface>
</hal>
```

A vendor must implement all of these instances:

android.hardware.vibrator.IVibrator/default   // version >= 1
android.hardware.vibrator.IVibrator/specific   // version >= 1
android.hardware.camera.ICamera/default   // version >= 5
android.hardware.camera.ICamera/${INSTANCE}
          // with version >= 5, where ${INSTANCE} matches [a-z]+/[0-9]+
          // e.g. legacy/0
Kernel matches
The <kernel> section of the framework compatibility matrix describes the framework's requirements of the Linux kernel on the device. This information is meant to be matched against the information about the kernel that gets reported by the device's VINTF object.

Match kernel branches
Each kernel branch suffix (for example, 5.4-r) is mapped to a unique kernel FCM version (for example, 5). The mapping is the same as the mapping between release letters (for example, R) and FCM versions (for example, 5).

VTS tests enforce that the device explicitly specifies the kernel FCM version in the device manifest, /vendor/etc/vintf/manifest.xml, if one of the following is true:

The kernel FCM version is different from the target FCM version. For example, the aforementioned device has a target FCM version 4, and its kernel FCM version is 5 (kernel branch suffix r).
The kernel FCM version is greater than or equal to 5 (kernel branch suffix r).
Note: If the kernel FCM version equals the target FCM version and they are greater than or equal to 5, the source file of the device manifest specified by DEVICE_MANIFEST_FILE doesn't have to specify the kernel FCM version. assemble_vintf automatically infers the kernel FCM version from the target FCM version if the target FCM version is greater than or equal to 5.
VTS tests enforce that, if the kernel FCM version is specified, the kernel FCM version is greater than or equal to the target FCM version in the device manifest.

Example: Determine the kernel branch
If the device has target FCM version 4 (released in Android 10), but runs kernel from the 4.19-r branch, the device manifest should specify the following:

```
<manifest version="2.0" type="device" target-level="4">
  <kernel target-level="5" />
</manifest>
```

The VINTF object checks kernel compatibility against requirements on 4.19-r kernel branch, which is specified in FCM version 5. These requirements are built from kernel/configs/r/android-4.19 in the Android source tree.

Example: Determine the kernel branch for GKI
If the device uses the Generic Kernel Image (GKI), and the kernel release string from /proc/version is the following:

```
5.4.42-android12-0-00544-ged21d463f856
```

Then, the VINTF object obtains the Android release from the kernel release, and use it to determine the kernel FCM version. In this example, android12 means kernel FCM version 6 (released in Android 12).

For details on how the kernel release string is parsed, see GKI versioning.

Match kernel versions
A matrix can include multiple <kernel> sections, each with a different version attribute using the format:

```
${ver}.${major_rev}.${kernel_minor_rev}
```

The VINTF object considers only the <kernel> section from the FCM with matching FCM version with the same ${ver} and ${major_rev} as the device kernel (that is, version="${ver}.${major_rev}.${matrix_minor_rev}"); other sections are ignored. In addition, the minor revision from the kernel must be a value from the compatibility matrix (${kernel_minor_rev} >= ${matrix_minor_rev};). If no <kernel> section meets these requirements, it's a non-match.

Example: Select requirements for matching
Consider the following hypothetical case where FCMs in /system/etc/vintf declare the following requirements (header and footer tags are omitted):

```
<!-- compatibility_matrix.3.xml -->
<kernel version="4.4.107" level="3"/>
```

```
<!-- See kernel/configs/p/android-4.4/ for 4.4-p requirements -->
<kernel version="4.9.84" level="3"/>
<!-- See kernel/configs/p/android-4.9/ for 4.9-p requirements -->
<kernel version="4.14.42" level="3"/>
<!-- See kernel/configs/p/android-4.14/ for 4.14-p requirements -->

<!-- compatibility_matrix.4.xml -->
<kernel version="4.9.165" level="4"/>
<!-- See kernel/configs/q/android-4.9/ for 4.9-q requirements -->
<kernel version="4.14.105" level="4"/>
<!-- See kernel/configs/q/android-4.14/ for 4.14-q requirements -->
<kernel version="4.19.42" level="4"/>
<!-- See kernel/configs/q/android-4.19/ for 4.19-q requirements -->

<!-- compatibility_matrix.5.xml -->
<kernel version="4.14.180" level="5"/>
<!-- See kernel/configs/r/android-4.14/ for 4.14-r requirements -->
<kernel version="4.19.123" level="5"/>
<!-- See kernel/configs/r/android-4.19/ for 4.19-r requirements -->
<kernel version="5.4.41" level="5"/>
<!-- See kernel/configs/r/android-5.4/ for 5.4-r requirements -->
```

The target FCM version, the kernel FCM version, and the kernel version together select the kernel requirements from the FCMs:

Target FCM version	Kernel FCM version	Kernel version	Match with
3 (P)	Unspecified	4.4.106	No match (minor version mismatch)
3 (P)	Unspecified	4.4.107	4.4-p
3 (P)	Unspecified	4.19.42	4.19-q (see note following the table)
3 (P)	Unspecified	5.4.41	5.4-r (see note following the table)
3 (P)	3 (P)	4.4.107	4.4-p
3 (P)	3 (P)	4.19.42	No match (no 4.19-p kernel branch)
3 (P)	4 (Q)	4.19.42	4.19-q
4 (Q)	Unspecified	4.4.107	No match (no 4.4-q kernel branch)
4 (Q)	Unspecified	4.9.165	4.9-q
4 (Q)	Unspecified	5.4.41	5.4-r (see note following the table)
4 (Q)	4 (Q)	4.9.165	4.9-q
4 (Q)	4 (Q)	5.4.41	No match (no 5.4-q kernel branch)
4 (Q)	5 (R)	4.14.105	4.14-r
4 (Q)	5 (R)	5.4.41	5.4-r
5 (R)	Unspecified	any	VTS fails (must specify the kernel FCM version for the target FCM version 5)
5 (R)	4 (Q)	any	VTS fails (kernel FCM version < target FCM version)
5 (R)	5 (R)	4.14.180	4.14-r

Note: For devices with a target FCM version lower than 5 (R) and an unspecified kernel FCM version, the VINTF object selects the kernel branch with the lowest suffix that is greater than the target FCM version with the matching kernel version. In the above example, a device with a target FCM version of 3 (P) use the 4.19-q kernel branch because 4.19-q has the lowest suffix among all 4.19-* kernel branches, [4.19-q, 4.19-r, 4.19-s]. This requirement selection behavior is disallowed for devices with a target FCM version of 5 (R) or higher, because the kernel FCM version must be specified.

Match kernel configurations

If the <kernel> section does match, the process continues by attempting to match config elements against /proc/config.gz. For each config element in the compatibility matrix, it looks up /proc/config.gz to see if the config is present. When a config item is set to n in the compatibility matrix for the matching <kernel> section, it must be absent from /proc/config.gz. Finally, a config item not in the compatibility matrix might or might not be present in /proc/config.gz.

Example: Match kernel configurations

<value type="string">bar</value> matches "bar". Quotes are omitted in the compatibility matrix but present in /proc/config.gz.
<value type="int">4096</value> matches 4096, 0x1000, or 0X1000.
<value type="int">0x1000</value> matches 4096, 0x1000, or 0X1000.
<value type="int">0X1000</value> matches 4096, 0x1000, or 0X1000.
<value type="tristate">y</value> matches y.
<value type="tristate">m</value> matches m.
<value type="tristate">n</value> means the config item must NOT exist in /proc/config.gz.
<value type="range">1-0x3</value> matches 1, 2, or 3, or hexadecimal equivalent.

Example: Successful kernel match

A framework compatibility matrix with FCM version 1 has the following kernel information:

```
<kernel version="4.14.42">
```

```
  <config>
    <key>CONFIG_TRI</key>
    <value type="tristate">y</value>
  </config>
  <config>
    <key>CONFIG_NOEXIST</key>
    <value type="tristate">n</value>
  </config>
  <config>
    <key>CONFIG_DEC</key>
    <value type="int">4096</value>
  </config>
  <config>
    <key>CONFIG_HEX</key>
    <value type="int">0XDEAD</value>
  </config>
  <config>
    <key>CONFIG_STR</key>
    <value type="string">str</value>
  </config>
  <config>
    <key>CONFIG_EMPTY</key>
    <value type="string"></value>
  </config>
</kernel>
```

The kernel branch is matched first. The kernel branch is specified in the device manifest in manifest.kernel.target-level, which defaults to manifest.level if the former isn't specified:

If the kernel branch in the device manifest is 1, the process proceeds to the next step and checks the kernel version.
If the kernel branch in the device manifest is 2, there's no match to the matrix. VINTF objects read the kernel requirements from the matrix at FCM version 2 instead.
Then, the kernel version is matched. If a device in uname() reports:

4.9.84 (no match to matrix unless there's a separate kernel section with <kernel version="4.9.x">, where x <= 84)
4.14.41 (no match to matrix, smaller than version)
4.14.42 (match to matrix)
4.14.43 (match to matrix)
4.1.22 (no match to matrix unless there's a separate kernel section with <kernel version="4.1.x">, where x <= 22)
After the appropriate <kernel> section is selected, for each <config> item with value other than n, the corresponding entry should be present in /proc/config.gz; for each <config> item with value n, the corresponding entry shouldn't be present in /proc/config.gz. The content of <value> should exactly match the text after the equal sign (including quotes), up to the newline character or #, with leading and trailing whitespace truncated.

The following kernel configuration is an example of a successful match:

```
# comments don't matter
CONFIG_TRI=y
# CONFIG_NOEXIST shouldn't exist
CONFIG_DEC = 4096 # trailing comments and whitespaces are fine
CONFIG_HEX=57005  # 0XDEAD == 57005
CONFIG_STR="str"
CONFIG_EMPTY=""   # empty string must have quotes
CONFIG_EXTRA="extra config items are fine too"
```

The following kernel configuration is an example of an unsuccessful match:

```
CONFIG_TRI="y"   # mismatch: quotes
CONFIG_NOEXIST=y # mismatch: CONFIG_NOEXIST exists
CONFIG_HEX=0x0   # mismatch; value doesn't match
CONFIG_DEC=""    # mismatch; type mismatch (expect int)
CONFIG_EMPTY=1   # mismatch; expects ""
# mismatch: CONFIG_STR is missing
```

SEPolicy matches
SEPolicy requires the following matches:

<sepolicy-version> defines a closed range of minor versions for every major version. The SEPolicy version reported by the device must fall within one of these ranges to be compatible with the framework. Match rules are similar to HAL versions; it is a match if the SEPolicy version is higher or equal to the minimum version for the range. The maximum version is purely informational.
<kernel-sepolicy-version>, that is, the Policy DB version, must be less than security_policyvers() reported by the device.
Example: Successful SEPolicy match
The framework compatibility matrix states the following SEPolicy information:

<sepolicy>
   <kernel-sepolicy-version>30</kernel-sepolicy-version>
   <sepolicy-version>25.0</sepolicy-version>
   <sepolicy-version>26.0-3</sepolicy-version>
</sepolicy>
On the device:

The value returned by security_policyvers() must be greater than or equal to 30. Otherwise it isn't a match. For example:
If a device returns 29, it isn't a match.
If a device returns 31, it's a match.
SEPolicy version must be one of 25.0-∞ or 26.0-∞. Otherwise it isn't a match. (The -3 after 26.0 is purely informational.)
AVB version matches
The AVB version contains a MAJOR version and MINOR version, with the format as MAJOR.MINOR (for example, 1.0, 2.1). For details, refer to Versioning and Compatibility. AVB version has the following system properties:

ro.boot.vbmeta.avb_version is the libavb version in bootloader.
ro.boot.avb_version is the libavb version in Android OS (init/fs_mgr).
The system property appears only when the corresponding libavb has been used to verify AVB metadata (and returns OK). It's absent if a verification failure occurred (or no verification occurred at all).

A compatibility match compares the following:

sysprop ro.boot.vbmeta.avb_version with avb.vbmeta-version from framework compatibility matrix:
ro.boot.vbmeta.avb_version.MAJOR == avb.vbmeta-version.MAJOR
ro.boot.vbmeta.avb_version.MINOR >= avb.vbmeta-version.MINOR
sysprop ro.boot.avb_version with avb.vbmeta-version from framework compatibility matrix:
ro.boot.avb_version.MAJOR == avb.vbmeta-version.MAJOR
ro.boot.avb_version.MINOR >= avb.vbmeta-version.MINOR
The bootloader or Android OS might contain two copies of libavb libraries, each with a different MAJOR version for upgrade devices and launch devices. In this case, the same unsigned system image can be shared but the final signed system images are different (with different avb.vbmeta-version):

Figure 1. AVB version matches (/system is P, all other partitions are O).

Figure 2. AVB version matches (all partitions are P).

Example: Successful AVB version match
The framework compatibility matrix states the following AVB information:

<avb>
   <vbmeta-version>2.1</vbmeta-version>
</avb>
On the device:

ro.boot.avb_version == 1.0 &&
ro.boot.vbmeta.avb_version == 2.1 mismatch

ro.boot.avb_version == 2.1 &&
ro.boot.vbmeta.avb_version == 3.0 mismatch

ro.boot.avb_version == 2.1 &&
ro.boot.vbmeta.avb_version == 2.3 match

ro.boot.avb_version == 2.3 &&
ro.boot.vbmeta.avb_version == 2.1 match
Match AVB version during OTA
Note: This section doesn't apply to devices updating to Android 11.
For devices launched with Android 9 or lower, when updating to Android 10, the AVB version requirements in the framework compatibility matrix are matched against the current AVB version on the device. If the AVB version has a major version upgrade during an OTA (for example, from 0.0 to 1.0), the VINTF check for compatibility in OTA doesn't reflect the compatibility after the OTA.

To mitigate the issue, an OEM can place a fake AVB version in the OTA package (compatibility.zip) to pass the check. To do so:

Cherry-pick the following CLs to the Android 9 source tree:
CL 732261
CL 732262
Define BOARD_OTA_FRAMEWORK_VBMETA_VERSION_OVERRIDE for the device. Its value should equal the AVB version before the OTA, that is, the AVB version of the device when it was launched.
Rebuild the OTA package.
These changes automatically place BOARD_OTA_FRAMEWORK_VBMETA_VERSION_OVERRIDE as compatibility-matrix.avb.vbmeta-version in the following files:

/system/compatibility_matrix.xml (which isn't used in Android 9) on the device
system_matrix.xml in compatibility.zip in the OTA package
These changes don't affect other framework compatibility matrixes, including /system/etc/vintf/compatibility_matrix.xml. After the OTA, the new value in /system/etc/vintf/compatibility_matrix.xml is used for compatibility checks instead.

VNDK version matches
The device compatibility matrix declares the required VNDK version in compatibility-matrix.vendor-ndk.version. If the device compatibility matrix doesn't have a <vendor-ndk> tag, no requirements are imposed, and it's always considered a match.

If the device compatibility matrix does have a <vendor-ndk> tag, a <vendor-ndk> entry with a matching <version> is looked up from the set of VNDK vendor snapshots that's provided by the framework in the framework manifest. If such an entry doesn't exist, there's no match.

If such an entry does exist, the set of libraries enumerated in the device compatibility matrix must be a subset of the set of libraries stated in the framework manifest; otherwise, the entry isn't considered a match.

As a special case, if no libraries are enumerated in the device compatibility matrix, the entry is always considered a match, because an empty set is a subset of any set.
Example: Successful VNDK version match
If the device compatibility matrix states the following requirement on VNDK:

```
<!-- Example Device Compatibility Matrix -->
<vendor-ndk>
  <version>27</version>
  <library>libjpeg.so</library>
  <library>libbase.so</library>
</vendor-ndk>
```

In the framework manifest, only the entry with version 27 is considered.

```
<!-- Framework Manifest Example A -->
<vendor-ndk>
  <version>27</version>
  <library>libjpeg.so</library>
  <library>libbase.so</library>
  <library>libfoo.so</library>
</vendor-ndk>
```

Example A is a match, because VNDK version 27 is in the framework manifest, and {libjpeg.so, libbase.so, libfoo.so} ⊇ {libjpeg.so, libbase.so}.

```
<!-- Framework Manifest Example B -->
<vendor-ndk>
  <version>26</version>
  <library>libjpeg.so</library>
  <library>libbase.so</library>
</vendor-ndk>
```

<vendor-ndk>
<version>27</version>
<library>libbase.so</library>
</vendor-ndk>
Example B isn't a match. Even though VNDK version 27 is in the framework manifest, libjpeg.so isn't supported by the framework in that snapshot. VNDK version 26 is ignored.

System SDK version matches
The device compatibility matrix declares a set of required system SDK version in compatibility-matrix.system-sdk.version. There's a match only if the set is a subset of provided system SDK versions as declared in manifest.system-sdk.version in the framework manifest.

As a special case, if no system SDK versions are enumerated in the device compatibility matrix, it's always considered a match, because empty set is a subset of any set.
Example: Successful system SDK version match
If the device compatibility matrix states the following requirement on System SDK:

<!-- Example Device Compatibility Matrix -->
<system-sdk>
<version>26</version>
<version>27</version>
</system-sdk>
Then, the framework must provide system SDK version 26 and 27 to match:

<!-- Framework Manifest Example A -->
<system-sdk>
<version>26</version>
<version>27</version>
</system-sdk>
Example A is a match:

<!-- Framework Manifest Example B -->
<system-sdk>
<version>26</version>
<version>27</version>
<version>28</version>
</system-sdk>
Example B is a match:

<!-- Framework Manifest Example C -->
<system-sdk>
<version>26</version>
</system-sdk>
Example C isn't a match, because system SDK version 27 isn't provided.

Let me explain all the **VINTF match rules** in simple terms, using **real-world examples** to help you understand:

🔧 VINTF Matching in Simple Words

Android uses **VINTF (Vendor Interface)** to make sure your **vendor image** (drivers, HALs, etc.) can **work with the Android framework** (system image). It checks that everything matches at:

- ✅ Build time
- 🔄 OTA (update) time
- 🚀 Boot time
- 🧪 VTS testing

---

1. 📘 Framework Compatibility Matrix Version Match

Rule:
`manifest.target-level == compatibility_matrix.level`

🧠 Meaning: The device manifest must specify the **same FCM version** as the framework matrix.

Example:

- Manifest:

```xml
<manifest target-level="3">
```

- Framework matrix:

```xml
<compatibility-matrix level="3">
```

✅ Match.
❌ If one has level 2 and the other level 3 — no match.

---

2. 📦 HAL Match (HIDL & AIDL)

Think of a HAL like a "driver" interface (e.g., camera, vibrator).

✅ HIDL HAL Match Example

Framework requires:

```xml
<hal>
  <name>android.hardware.vibrator</name>
  <version>1-2</version>
  <interface>
    <name>IVibrator</name>
    <instance>default</instance>
    <instance>specific</instance>
  </interface>
</hal>
```

Your device must implement:

- `IVibrator/default` with version ≥ 1
- `IVibrator/specific` with version ≥ 1

✅ AIDL HAL Match Example

AIDL is newer (no major.minor, just version numbers like 5).

Matrix says:

```xml
<hal>
  <name>android.hardware.camera</name>
  <version>5</version>
</hal>
```

Device:

- Can implement version 5, 6, or higher — all OK.

---

3. 🧠 Kernel Match

The kernel must match in 3 ways:

1. **Kernel FCM version**

2. **Kernel version** (like 4.14.180)
3. **Kernel configs** (like `CONFIG_ABC=y`)

✅ Kernel Branch Match

Device:

```xml
<manifest target-level="4">
  <kernel target-level="5"/>
</manifest>
```

Kernel version: `4.19.123`
Matrix (level 5) has:

```xml
<kernel version="4.19.123"/>
```

✅ Match.

---

4. ⚙️ Kernel Config Match

From `/proc/config.gz`, check if configs match.

Matrix says:

```xml
<key>CONFIG_TRI</key> <value type="tristate">y</value>
<key>CONFIG_NOEXIST</key> <value type="tristate">n</value>
```

Device config:

```ini
CONFIG_TRI=y
# CONFIG_NOEXIST not present ✅
```

✅ Match
❌ If `CONFIG_NOEXIST=y` exists, it's a mismatch.

---

5. 🔐 SEPolicy Match

Checks SELinux policy compatibility.

Matrix says:

```xml
<kernel-sepolicy-version>30</kernel-sepolicy-version>
<sepolicy-version>26.0-3</sepolicy-version>
```

Device:

- `security_policyvers()` must return ≥ 30
- sepolicy version must be 26.0 or higher

✅ If device returns version 31 → match
❌ If device returns version 29 → no match

---

6. 🛡️ AVB Version Match (Verified Boot)

Matrix:

```xml
<vbmeta-version>2.1</vbmeta-version>
```

Device properties:

- **ro.boot.vbmeta.avb_version** = 2.3
- **ro.boot.avb_version** = 2.1

✅ Match (both major = 2, minor ≥ 1)

❌ Mismatch if major ≠ 2

---

7. 🧩 VNDK Version Match

Matrix says:

```xml
<vendor-ndk>
  <version>27</version>
  <library>libbase.so</library>
  <library>libjpeg.so</library>
</vendor-ndk>
```

Framework must list all those libraries in its manifest with version 27.

✅ Match if framework has all
❌ Mismatch if even 1 library (e.g., **libjpeg.so**) is missing

---

8. 📚 System SDK Match

Matrix:

```xml
<system-sdk>
  <version>26</version>
  <version>27</version>
</system-sdk>
```

Framework must provide both 26 and 27 in its manifest.

✅ If framework gives 26, 27, 28 → Match
❌ If it gives only 26 → No match

🧠 Summary Table

Component	What is Checked	Must Match
FCM Version	Manifest **target-level**	= matrix **level**
HAL	Interface version and instances	Follow OR/AND rules
Kernel	Version, Branch, Config	Version range and exact config
SEPolicy	Kernel & SEPolicy version	≥ specified
AVB	Bootloader + Android AVB versions	Major =, Minor ≥
VNDK	lib list	Matrix ⊆ Framework
System SDK	versions	Matrix ⊆ Framework

Let me know if you want a visual chart or need examples using real device XMLs.

12. can you explain match rules in simple words with example

Absolutely! Let's break down the **VINTF Match Rules** into simple words and examples — like you're explaining it to someone setting up Android for the first time.

✅ What are VINTF Match Rules?

When Android boots or updates, it **checks that the vendor image (drivers, HALs, kernel)** is compatible with the **system image (Android framework)** using **match rules**.

Think of it like a checklist that ensures:

> "Do both sides agree on what features and versions they support?"

🧩 1. FCM Version Match

🔧 Rule:

The **`target-level`** in your device's manifest must match the **`level`** in the framework compatibility matrix.

✅ Example:

Device manifest

```xml
<manifest target-level="3">
```

Framework matrix

```xml
<compatibility-matrix level="3">
```

✅ **Match!**

If one is `2` and the other `3` → ❌ No match.

📦 2. HAL Match

> HAL = Hardware Abstraction Layer, like camera, audio, etc.

🔧 Rule:

Each HAL in the **matrix** must be **present in the manifest** with compatible versions and instances.

✅ Example A: Basic match

Matrix:

```xml
<hal>
  <name>android.hardware.camera</name>
  <version>3.4</version>
  <interface>
    <name>ICamera</name>
    <instance>default</instance>
  </interface>
</hal>
```

Manifest:

```xml
<hal>
  <name>android.hardware.camera</name>
  <version>3.4</version>
  <interface>
    <name>ICamera</name>
    <instance>default</instance>
  </interface>
</hal>
```

✅ **Perfect match**

✅ Example B: Higher version allowed

Matrix:

```xml
<version>3.1-5</version>  <!-- Accepts versions from 3.1 to 5 -->
```

Manifest:

```xml
<version>3.4</version>
```

✅ **Still compatible**
(Because 3.4 is in range `3.1–5`)

❌ Wrong version

Matrix: Wants version 3.4
Manifest: Gives version 3.0

❌ **Not compatible** – too old.

🧠 3. Kernel Match

Checks the kernel's:

- Version (e.g., `5.4.41`)
- Config items (`CONFIG_XYZ=y`)
- Kernel branch (e.g., `android12-5.4`)

✅ Example:

Matrix requires:

```xml
<kernel version="5.4.41">
  <config>
    <key>CONFIG_64BIT</key>
    <value type="tristate">y</value>
  </config>
</kernel>
```

Device has:

```ini
CONFIG_64BIT=y
```

✅ **Match**

❌ Mismatch example:

Matrix says:

```xml
<key>CONFIG_DEBUG</key>
<value type="tristate">n</value>
```

But device has:

```ini
CONFIG_DEBUG=y
```

❌ **No match** (Because `n` means the config must **not exist**)

🔐 4. SEPolicy Match (Security Policies)

Rule:

- `kernel-sepolicy-version` from matrix ≤ value from device
- `sepolicy-version` in manifest must be ≥ required version

✅ Example:

Matrix:

```xml
<kernel-sepolicy-version>30</kernel-sepolicy-version>
<sepolicy-version>26.0-3</sepolicy-version>
```

Device:

- `security_policyvers()` = `31`
- `sepolicy-version` = `26.1`

✅ Match (31 ≥ 30, and 26.1 is allowed)

🔒 5. AVB Version Match (Verified Boot)

Checks `avb_version` (boot verification).

✅ Match Example:

Matrix:

```xml
<vbmeta-version>2.1</vbmeta-version>
```

Device:

- `ro.boot.avb_version = 2.3`
- `ro.boot.vbmeta.avb_version = 2.1`

✅ **Match** (2.3 ≥ 2.1)

📚 6. VNDK Version Match

Vendor NDK libraries must match the system's supported version.

✅ Example:

Device Compatibility Matrix:

```xml
<vendor-ndk>
  <version>27</version>
  <library>libbase.so</library>
</vendor-ndk>
```

Framework Manifest:

```xml
<vendor-ndk>
  <version>27</version>
  <library>libbase.so</library>
  <library>libfoo.so</library>
</vendor-ndk>
```

✅ Match (device's libs are subset of system)

📦 7. System SDK Match

Framework APIs used by vendor must be supported by system.

✅ Example:

Device Compatibility Matrix:

```xml
<system-sdk>
  <version>27</version>
  <version>28</version>
</system-sdk>
```

Framework Manifest:

```xml
<system-sdk>
  <version>26</version>
  <version>27</version>
  <version>28</version>
</system-sdk>
```

✅ **Match**

🧠 Summary Table (Visual)

Check Type	Must Match	OK If...
FCM Version	`manifest.target-level == matrix.level`	Exact match
HALs	Versions & instances	Manifest version is ≥ matrix min
Kernel	Version, branch, configs	Kernel matches version & configs
SEPolicy	Policy versions	Device has ≥ required
AVB	avb version (bootloader + Android)	MAJOR ==, MINOR ≥
VNDK	Library list	Device libs ⊆ system libs
System SDK	Versions	Device ⊆ system SDK

Let me know if you want a **diagram**, actual **XML samples**, or **validation scripts** for these.

13. Additional resources

bookmark_border
The following resources provide details on code locations, tools, testing, and licensing.

Queryable code location
The code for the queryable vendor interface object goes to system/libvintf.

Tools
Handwriting manifest files and compatibility matrixes can be tough. Use the following tools to generate a boilerplate manifest/compatibility matrix to start from.

LSHAL
LSHAL is a device-side tool that lists all registered HALs to hwservicemanager and all available passthrough implementations (e.g. android.hardware.foo@1.0-impl.so) on the device. It can also generate a device manifest file based on the list:

```
adb shell su 0 /system/bin/lshal --init-vintf
```

Note the following:

If a package is both registered to hwservicemanager and found as a passthrough HAL, <transport> is set to hwbinder.
No SELinux version is written into the manifest. It is suggested that the element is injected through assemble_vintf as explained below.
The generated HAL manifest file may be inaccurate. Human attention is required to fix inconsistencies between the device manifest and what vendor.img actually provides.
ASSEMBLE_VINTF
assemble_vintf is a host-side tool that:

Verifies a compatibility matrix or manifest file is valid.
Injects variables to manifests/compatibility matrixes available at build time and generates a new file that should be installed to the device.
Checks compatibility between the generated file and its dual.
If a manifest file is given, optionally generates a boilerplate compatibility matrix that is compatible with the manifest file.
Example: Generate device compatibility matrix from a framework manifest file

```
assemble_vintf -m --hals-only \
    -i system/libhidl/manifest.xml \
    -o device/manufacturer/device_name/compatibility_matrix.xml
```

Note that all HALs are set to optional="true".

Example: Generate a skeleton framework compatibility matrix from a device manifest file

```
assemble_vintf -m --hals-only \
    -i device/foo/bar/manifest.xml \
    -o path/to/place/output/compatibility_matrix.xml
```

Note that all HALs are set to optional="true".

Example: Generate device manifest XML files from variables
At build time, if the following variables are defined in device/manufacturer/device_name/BoardConfig.mk:

```
# Vendor manifest is named DEVICE_MANIFEST_FILE for legacy reasons.
DEVICE_MANIFEST_FILE := \
    device/manufacturer/device_name/vendor_manifest.xml
ODM_MANIFEST_FILES := \
    device/manufacturer/device_name/odm_manifest.xml
ODM_MANIFEST_SKUS := sku1 sku2
ODM_MANIFEST_SKU1_FILES := \
    device/manufacturer/device_name/odm_manifest_sku1.xml
ODM_MANIFEST_SKU2_FILES := \
    device/manufacturer/device_name/odm_manifest_sku2.xml
```

Then the following commands are executed (in the build system, modified to omit implementation details) to generate device manifest XML files:

```
# vendor manifest; only when DEVICE_MANIFEST_FILE is set
```

```
BOARD_SEPOLICY_VERS=$(BOARD_SEPOLICY_VERS) assemble_vintf \
    $(addprefix,-i ,$(DEVICE_MANIFEST_FILE)) \
    -o $(TARGET_OUT_VENDOR)/etc/vintf/manifest.xml

# ODM manifests
assemble_vintf \
    $(addprefix,-i ,$(ODM_MANIFEST_FILES)) \
    -o $(TARGET_OUT_ODM)/etc/vintf/manifest.xml

# ODM manifests for each sku
assemble_vintf \
    $(addprefix,-i ,$(ODM_MANIFEST_SKU1_FILES)) \
    -o $(TARGET_OUT_ODM)/etc/vintf/manifest_sku1.xml
assemble_vintf \
    $(addprefix,-i ,$(ODM_MANIFEST_SKU2_FILES)) \
    -o $(TARGET_OUT_ODM)/etc/vintf/manifest_sku2.xml
```

At runtime, the VINTF object combines vendor manifests and ODM manifests as the device manifest. See Device manifest for details.

Example: Generate device compatibility matrix XML files from variables

At build time, if the following variables are defined in device/manufacturer/device_name/BoardConfig.mk:

```
# vendor compatibility matrix is named DEVICE_MATRIX_FILE for legacy reasons.
DEVICE_MATRIX_FILE := \
    device/manufacturer/device_name/vendor_compatibility_matrix.xml \
    device/manufacturer/device_name/vendor_compatibility_matrix_additional.xml
```

Then the following commands are executed (in the build system, modified to omit implementation details) to generate device compatibility matrix XML files:

```
# vendor compatibility matrix; only when DEVICE_MATRIX_FILE is set
assemble_vintf \
    $(addprefix,-i ,$(DEVICE_MATRIX_FILE)) \
    -o $(TARGET_OUT_VENDOR)/etc/vintf/compatibility_matrix.xml
```

At runtime, the VINTF object uses the vendor compatibility matrix as the device compatibility matrix. See Device compatibility matrix for details.

Example: Generate framework manifest XML files from variables

The following variables may be defined in device/manufacturer/device_name/BoardConfig.mk:

```
# Device-specific system manifest is named DEVICE_FRAMEWORK_MANIFEST_FILE for legacy reasons
DEVICE_FRAMEWORK_MANIFEST_FILE := \
    device/manufacturer/device_name/device_system_manifest.xml

# Product manifest
PRODUCT_MANIFEST_FILES := \
    device/manufacturer/device_name/product_manifest.xml
```

The following commands are executed (in the build system, modified to omit implementation details) to generate framework manifest XML files:

```
# system manifest
assemble_vintf \
    -i system/libhidl/vintfdata/manifest.xml \
    $(addprefix,-i ,$(DEVICE_FRAMEWORK_MANIFEST_FILE)) \
    -o $(TARGET_OUT)/etc/vintf/manifest.xml

# product manifest
assemble_vintf \
    $(addprefix,-i ,$(PRODUCT_MANIFEST_FILES)) \
    -o $(TARGET_OUT_PRODUCT)/etc/vintf/manifest.xml
```

At runtime, the VINTF object combines the system manifest, system manifest fragments, product manifest, and product manifest fragments as the framework manifest. See Framework manifest for details.

Example: Generate framework compatibility matrix XML files from variables

The following variables may be defined in device/manufacturer/device_name/BoardConfig.mk to define the product FCM and device-specific system FCM:

```
DEVICE_PRODUCT_COMPATIBILITY_MATRIX_FILE := \
    device/manufacturer/device_name/product_compatibility_matrix.xml
# Device-specific system compatibility matrix is named
# DEVICE_FRAMEWORK_COMPATIBILITY_MATRIX_FILE for legacy reasons.
DEVICE_FRAMEWORK_COMPATIBILITY_MATRIX_FILE := \
    device/manufacturer/device_name/device_system_compatibility_matrix.xml
```

The system_ext FCM must be installed with Soong modules. The product FCM may also be installed with Soong modules; do not define DEVICE_PRODUCT_COMPATIBILITY_MATRIX_FILE if this method is used. In addition, multiple product FCM versions and system_ext FCM versions may be installed with Soong modules. Define the following:

Define a module in device/manufacturer/device_name/Android.bp. For example (replace system_ext with product for product FCM):

```
vintf_compatibility_matrix {
    name: "system_ext_compatibility_matrix.xml",
    stem: "compatibility_matrix.xml",
    system_ext_specific: true,
    // product_specific: true, // for product FCM
    srcs: [
        "system_ext_compatibility_matrix.xml",
    ],
}
```

Install the module to device/manufacturer/device_name/device.mk. For example:

```
PRODUCT_PACKAGES += system_ext_compatibility_matrix.xml
```

The following commands are executed (in the build system, modified to omit implementation details) to generate framework compatibility matrix XML files:

```
# common system compatibility matrix for each FCM version
BOARD_SEPOLICY_VERS=$(BOARD_SEPOLICY_VERS) \
POLICYVERS=$(POLICYVERS) \
BOARD_AVB_VBMETA_VERSION=$(BOARD_AVB_VBMETA_VERSION)
assemble_vintf \
    -i hardware/interfaces/compatibility_matrices/compatibility_matrix.empty.xml
    $(addprefix,-i ,$(DEVICE_FRAMEWORK_COMPATIBILITY_MATRIX_FILE)) \
    -o $(TARGET_OUT)/etc/vintf/compatibility_matrix.device.xml

# framework compatibility matrixes at each FCM version
assemble_vintf
    -i hardware/interfaces/compatibility_matrices/compatibility_matrix.{level}.xml \
    -o $(TARGET_OUT)/etc/vintf/compatibility_matrix.{level}.xml \
    --kernel=...

# product framework compatibility matrix; only when
# DEVICE_PRODUCT_COMPATIBILITY_MATRIX_FILE is set or when the Soong module for
# product FCM is defined
assemble_vintf
    -i $(DEVICE_PRODUCT_COMPATIBILITY_MATRIX_FILE)
    -o $(TARGET_OUT_PRODUCT)/etc/vintf/compatibility_matrix.xml

# system_ext framework compatibility matrix; only when the Soong module for
# system_ext FCM is defined
assemble_vintf
    -i <srcs for the soong module>
    -o $(TARGET_OUT_SYSTEM_EXT)/etc/vintf/compatibility_matrix.xml
```

At runtime, the VINTF object combines a subset of system compatibility matrixes and product compatibility matrixes as the framework compatibility matrix. See Framework compatibility matrix for details.

Example: Generate the vendor manifest from fragments

Multiple vendor manifest fragments can be bundled at build time. For example:

```
<!-- device/manufacturer/device_name/manifest_common.xml -->
<manifest version="1.0" type="device">
```

```
    <!-- common HALs here -->
</manifest>

<!-- device/manufacturer/device_name/ir.xml -->
<manifest version="1.0" type="device">
    <hal>
        <name>android.hardware.ir</name>
        <version>1.0</version>
        <!-- other fields -->
    </hal>
</manifest>

# device/manufacturer/device_name/BoardConfig.mk
DEVICE_MANIFEST_FILE := device/manufacturer/device_name/manifest_common.xml
ifdef BOARD_ENABLE_IR
    DEVICE_MANIFEST_FILE += device/manufacturer/device_name/ir.xml
endif
Then, assemble_vintf adds the IR HAL to the vendor manifest if BOARD_ENABLE_IR is defined, and omits it if BOARD_ENABLE_IR is not defined.
The following commands (modified to omit implementation details) are executed to generate the vendor manifest:


# if BOARD_ENABLE_IR is defined
BOARD_SEPOLICY_VERS=10000.0 assemble_vintf \
    -i device/manufacturer/device_name/manifest_common.xml:device/manufacturer/device_name/ir.xml \
    -o $(TARGET_OUT_VENDOR)/manifest.xml

# if BOARD_ENABLE_IR is not defined
BOARD_SEPOLICY_VERS=10000.0 assemble_vintf \
    -i device/manufacturer/device_name/manifest_common.xml \
    -o $(TARGET_OUT_VENDOR)/manifest.xml
For details, see:


assemble_vintf --help
Testing
The platform/system/libvintf project uses GTest for the serialization, deserialization, and compatibility checking.

Licensing
tinyxml2 (external/tinyxml2) for serializing/deserializing the object to/from XML. BSD-like license.
libselinux (external/selinux/libselinux) for getting policydb version. Public domain license.
libz (external/zlib) for decompressing /proc/config.gz. BSD-like license.
libvintf project uses Apache 2.0 license (with appropriate MODULE_LICENSE_APACHE2 and NOTICE files).
```

Great! You've pasted a full reference guide for `assemble_vintf` and VINTF-related tools. Let me now **explain this entire thing in simple words with clear structure and examples**:

📘 Overview

When you're building Android for a new device, you need to declare **what hardware/software your device supports** and make sure it's **compatible** with the Android version you are building.

This is managed using:

- **Manifests** (`manifest.xml`) → Declare *what the device or framework provides*
- **Compatibility Matrices** (`compatibility_matrix.xml`) → Declare *what the system or vendor expects*

🧰 Tools

1. lshal (device-side)

What it does:

- Lists all HALs currently in use on the device

- Can auto-generate a manifest template

🔧 Example:

```sh
adb shell su 0 /system/bin/lshal --init-vintf
```

🔹 This generates a basic **`manifest.xml`** by scanning HALs.

---

2. assemble_vintf (host-side build tool)

What it does:

✅ Validates manifest or matrix
✅ Fills in runtime values like **`sepolicy-version`**, **`avb_version`**, etc.
✅ Generates combined or final XML files for your device
✅ Checks compatibility between two files

🧪 Common Use Cases & Simple Examples

✅ 1. Generate Device Compatibility Matrix from Framework Manifest

When?

If you're building a vendor partition and want to know what the system expects.

Command:

```sh
assemble_vintf -m --hals-only \
  -i system/libhidl/manifest.xml \
  -o device/<vendor>/<product>/compatibility_matrix.xml
```

🔸 Output: All HALs marked **`optional=true`** so you can fine-tune later.

✅ 2. Generate Framework Compatibility Matrix from Device Manifest

When?

If you're building a system image to work with an existing vendor partition.

Command:

```sh
assemble_vintf -m --hals-only \
  -i device/<vendor>/<product>/manifest.xml \
  -o path/to/output/compatibility_matrix.xml
```

✅ 3. Use BoardConfig.mk to Define XML Paths

You define paths in your `BoardConfig.mk`:

```makefile
DEVICE_MANIFEST_FILE := device/samsung/galaxy/manifest.xml
DEVICE_MATRIX_FILE := device/samsung/galaxy/compatibility_matrix.xml
```

Then build system runs:

```sh
assemble_vintf -i $(DEVICE_MANIFEST_FILE) -o $(TARGET_OUT_VENDOR)/etc/vintf/manifest.xml
assemble_vintf -i $(DEVICE_MATRIX_FILE) -o $(TARGET_OUT_VENDOR)/etc/vintf/compatibility_matrix.xml
```

✅ 4. Multiple Fragments Based on SKU

Imagine:

- You want to include **IR HAL** only if **IR sensor exists**

Setup:

manifest_common.xml:

```xml
<manifest version="1.0" type="device">
  <!-- common HALs -->
</manifest>
```

ir.xml:

```xml
<hal>
  <name>android.hardware.ir</name>
  <version>1.0</version>
</hal>
```

`BoardConfig.mk`:

```makefile
DEVICE_MANIFEST_FILE := device/xyz/device/manifest_common.xml
ifdef BOARD_ENABLE_IR
  DEVICE_MANIFEST_FILE += device/xyz/device/ir.xml
endif
```

Then:

```sh
assemble_vintf -i manifest_common.xml:ir.xml -o manifest.xml
```

👉 Adds IR HAL only when **`BOARD_ENABLE_IR`** is defined

📁 Where Files End Up

File Type	Final Output Location
Vendor manifest	**`$(TARGET_OUT_VENDOR)/etc/vintf/manifest.xml`**
ODM manifest	**`$(TARGET_OUT_ODM)/etc/vintf/manifest.xml`**
System manifest	**`$(TARGET_OUT)/etc/vintf/manifest.xml`**
Product manifest	**`$(TARGET_OUT_PRODUCT)/etc/vintf/manifest.xml`**
Framework compatibility	**`$(TARGET_OUT)/etc/vintf/compatibility_matrix.device.xml`**
Product compatibility	**`$(TARGET_OUT_PRODUCT)/etc/vintf/compatibility_matrix.xml`**
System_ext compatibility	**`$(TARGET_OUT_SYSTEM_EXT)/etc/vintf/compatibility_matrix.xml`**

🧪 Testing

The tool internally uses **GTest** to:

- Parse and validate XML files

- Compare version match rules
- Ensure **`manifest.xml`** and **`compatibility_matrix.xml`** are compatible

📄 Licenses & Libraries Used

Component	Library Used	License
XML parsing	**`tinyxml2`**	BSD-like
SELinux version	**`libselinux`**	Public domain
Kernel config.gz	**`zlib`**	BSD-like
Main project	**`libvintf`**	Apache 2.0

✅ Summary in 1 Sentence:

> Use **`lshal`** on the device to generate HAL info, and **`assemble_vintf`** during build time to validate and merge HAL/SEPolicy/AVB/etc. configs into final **`manifest.xml`** and **`compatibility_matrix.xml`**.

Would you like a visual **flow diagram** of how these files are generated and merged at runtime?