HAL-HIDL

1. Hardware abstraction layer (HAL) overview

bookmark_border
A hardware abstraction layer (HAL) is type of abstraction layer with a standard interface for hardware vendors to implement. A HAL allows hardware vendors to implement lower-level, device-specific features without affecting or modifying code in higher-level layers.

Note: HALs existed before Android 8. However, Android 8 ensured each HAL had a standard interface. For an explanation of HAL architecture before Android 8, refer to Legacy HALs.
Required terms
Following is a list of definitions for terms used in this section of documentation:

Android Interface Definition Language (AIDL)
A Java-like language used to define interfaces in a way that is independent of the programming language being used. AIDL allows communication between HAL clients and HAL services.
Binderized HAL
A HAL that communicates with other processes using binder inter-process communication (IPC) calls. Binderized HALs run in a separate process from the client that uses them. Binderized HALs are registered with a service manager so that clients can access their capabilities. HALs written for Android 8 and higher are binderized.
HAL client
The process that accesses the HAL service.
HAL interface
The common interface used by both the HAL client and service to communicate.
HAL service
The hardware-specific code, such as the code that talks to your specific device's camera. You must implement all required HALs listed in the compatibility matrix for the release you target in your vendor partition. For further information on compatibility matrixes, see Compatibility matrixes.

You can also create a new HAL interface, or extend the interface of a HAL, to support your hardware's unique capabilities.

Hardware Interface Definition Language (HIDL)
A language used to define interfaces in a way that is independent of the programming language being used. HIDL enables communication between HAL clients and HAL services.

Note: As of Android 13, HIDL has been deprecated. Instead of HIDL, you should use Android Interface Definition Language (AIDL) for HALs. HALs previously written using HIDL are supported.
Same Process (SP) HAL
A restricted set of wrapped HALs. Membership in the set is controlled by Google. SP HALs include:

Stable C mapper 5 HAL
OpenGL
Vulkan
android.hidl.memory@1.0 (provided by the Android system, always wrapped)
android.hardware.renderscript@1.0
Note: Some HAL implementations wrap legacy interfaces, such as those in libhardware, in an AIDL or HIDL wrapper.
Wrapped HAL
A HAL service that was created before Android 8, but has been wrapped in an AIDL or HIDL wrapper to work with Android 8 and higher.

What's next?
To learn how to implement a HAL service, refer to AIDL for HALs.
To learn how to create or extend an existing HAL, refer to Attached extended interfaces.

Here's a **simple explanation of HAL (Hardware Abstraction Layer)** in Android:

🔧 What is HAL?

The **Hardware Abstraction Layer (HAL)** is a *middle layer* that sits between Android's higher-level system/framework (like Camera API or Audio Manager) and your actual hardware (like camera sensor or speaker).

It allows Android to talk to **hardware in a consistent way**, even if the actual hardware differs between devices.

🤝 Why do we need HAL?

HAL provides a **standard interface** that hardware vendors (like Qualcomm, MediaTek, etc.) can implement. This lets Android work the same way across devices without needing to know the details of every piece of hardware.

Example:

- Android calls `Camera.open()` → framework talks to HAL → HAL talks to the actual camera driver.

📜 Important Terms:

Term	Meaning
HAL client	The system/framework code that wants to use the hardware (e.g., Camera app, Audio service).
HAL service	The vendor-specific code that controls the actual hardware.
HAL interface	The communication format or contract between the client and service.
Binderized HAL	HAL runs in its own process and communicates using Binder IPC (recommended since Android 8).
Same Process (SP) HAL	Special HALs that run in the same process as the framework for performance (Google-controlled).
Wrapped HAL	Old-style HALs wrapped with new AIDL/HIDL interface to work with Android 8+.
AIDL (Android Interface Definition Language)	Used to define new HALs now. Works across processes, like a remote function call.
HIDL (Hardware Interface Definition Language)	Was used for HALs from Android 8 to 12. Now deprecated in favor of AIDL.

📅 Timeline

- **Before Android 8**: HALs were custom C/C++ interfaces (no standardization).
- **Android 8 to 12**: HALs had to use **HIDL**.
- **Android 13+**: New HALs must use **AIDL**. HIDL HALs are still supported.

🔜 What to learn next?

- Learn how to **implement an AIDL-based HAL** → see Android docs on "AIDL for HALs".
- Learn how to **extend or create HAL interfaces** → see "Attached extended interfaces".

💡 Simple Analogy:

Imagine Android as a universal **remote control**.

- The **HAL** is the IR code translator that understands how to talk to **each different brand** of TV (hardware).
- The **framework** just sends "increase volume" — HAL makes sure the correct signal is sent to your Sony, Samsung, or LG TV.

Let me know if you want to see:

- 🔧 A real AIDL HAL code example,
- 🏗️ Full flow: from AIDL file to service implementation,
- 🧠 HIDL vs AIDL comparison in-depth.

2. HIDL

bookmark_border
HAL interface definition language or HIDL is an interface description language (IDL) to specify the interface between a HAL and its users. HIDL allows specifying types and method calls, collected into interfaces and packages. More broadly, HIDL is a system for communicating between codebases that may be compiled independently.

Caution: As of Android 10, HIDL is deprecated and has been replaced By Android Interface Definition Language (AIDL).
HIDL is intended to be used for inter-process communication (IPC). HALs created with HDL are called binderized HALs in that they can communicate with other architecture layers using binder inter-process communication (IPC) calls. Binderized HALs run in a separate process from the client that uses them. For libraries that must be linked to a process, a passthrough mode is also available (not supported in Java).

HIDL specifies data structures and method signatures, organized in interfaces (similar to a class) that are collected into packages. The syntax of HIDL looks familiar to C++ and Java programmers, but with a different set of keywords. HIDL also uses Java-style annotations.

Terminology
This section uses the following HIDL-related terms:

binderized Indicates HIDL is being used for remote procedure calls between processes, implemented over a Binder-like mechanism. See also passthrough.
callback, asynchronous Interface served by a HAL user, passed to the HAL (using a HIDL method), and called by the HAL to return data at any time.
callback, synchronous Returns data from a server's HIDL method implementation to the client. Unused for methods that return void or a single primitive value.
client Process that calls methods of a particular interface. A HAL or Android framework process may be a client of one interface and a server of another. See also passthrough.
extends Indicates an interface that adds methods and/or types to another interface. An interface can extend only one other interface. Can be used for a minor version increment in the same package name or for a new package (e.g. a vendor extension) to build on an older package.
generates Indicates an interface method that returns values to the client. To return one non-primitive value, or more than one value, a synchronous callback function is generated.
interface Collection of methods and types. Translated into a class in C++ or Java. All methods in an interface are called in the same direction: a client process invokes methods implemented by a server process.
oneway When applied to a HIDL method, indicates the method returns no values and doesn't block.
package Collection of interfaces and data types sharing a version.
passthrough Mode of HIDL in which the server is a shared library, dlopened by the client. In passthrough mode, client and server are the same process but separate codebases. Used only to bring legacy codebases into the HIDL model. See also Binderized.
server Process that implements methods of an interface. See also passthrough.
transport HIDL infrastructure that moves data between the server and client.
version Version of a package. Consists of two integers, major and minor. Minor version increments may add (but not change) types and methods.
HIDL design
The goal of HIDL is that the Android framework can be replaced without having to rebuild HALs. HALs are built by vendors or SOC makers and put in a /vendor partition on the device, enabling the Android framework, in its own partition, to be replaced with an OTA without recompiling the HALs.

HIDL design balances the following concerns:

Interoperability. Create reliably interoperable interfaces between processes which may be compiled with various architectures, toolchains, and build configurations. HIDL interfaces are versioned and cannot be changed after they are published.
Efficiency. HIDL tries to minimize the number of copy operations. HIDL-defined data is delivered to C++ code in C++ standard layout data structures that can be used without unpacking. HIDL also provides shared memory interfaces and, as RPCs are inherently somewhat slow, HIDL supports two ways to transfer data without using an RPC call: shared memory and a Fast Message Queue (FMQ).
Intuitive. HIDL avoids thorny issues of memory ownership by using only in parameters for RPC (see Android Interface Definition Language (AIDL)); values that cannot be efficiently returned from methods are returned via callback functions. Neither passing data into HIDL for transfer nor receiving data from HIDL changes the ownership of the data—ownership always remains with the calling function. Data needs to persist only for the duration of the called function and may be destroyed immediately after the called function returns.
Use passthrough mode
To update devices running earlier versions of Android to Android O, you can wrap both conventional (and legacy) HALs in a new HIDL interface that serves the HAL in binderized and same-process (passthrough) modes. This wrapping is transparent to both the HAL and the Android framework.

Passthrough mode is available only for C++ clients and implementations. Devices running earlier versions of Android don't have HALs written in Java, so Java HALs are inherently binderized.

Passthrough header files
When a .hal file is compiled, hidl-gen produces an extra passthrough header file BsFoo.h in addition to the headers used for binder communication; this header defines functions to be dlopened. As passthrough HALs run in the same process in which they are called, in most cases passthrough methods are invoked by direct function call (same thread). oneway methods run in their own thread as they aren't intended to wait for the HAL to process them (this means any HAL that uses oneway methods in passthrough mode must be thread-safe).

Given an IFoo.hal, BsFoo.h wraps the HIDL-generated methods to provide additional features (such as making oneway transactions run in another thread). This file is similar to BpFoo.h, however instead of passing on calls IPC using binder, the desired functions are directly invoked. Future implementations of HALs may provide multiple implementations, such as FooFast HAL and a FooAccurate HAL. In such cases, a file for each additional implementation would be created (e.g., PTFooFast.cpp and PTFooAccurate.cpp).

Binderizing passthrough HALs
You can binderize HAL implementations that support passthrough mode. Given a HAL interface a.b.c.d@M.N::IFoo, two packages are created:

a.b.c.d@M.N::IFoo-impl. Contains the implementation of the HAL and exposes function IFoo* HIDL_FETCH_IFoo(const char* name). On legacy devices, this package is dlopened and the implementation is instantiated using HIDL_FETCH_IFoo. You can generate the base code using hidl-gen and -Lc++-impl and -Landroidbp-impl.
a.b.c.d@M.N::IFoo-service. Opens the passthrough HAL and registers itself as a binderized service, enabling the same HAL implementation to be used as both passthrough and binderized.
Given the type IFoo, you can call sp<IFoo> IFoo::getService(string name, bool getStub) to get access to an instance of IFoo. If getStub is true, getService attempts to open the HAL only in passthrough mode. If getStub is false, getService attempts to find a binderized service; if that fails, it then tries to find the passthrough service. The getStub parameter should never be used except in defaultPassthroughServiceImplementation. (Devices launching with Android O are fully binderized devices, so opening a service in passthrough mode is disallowed.)

HIDL grammar
By design, the HIDL language is similar to C (but doesn't use the C preprocessor). All punctuation not described below (aside from the obvious use of = and |) is part of the grammar.

Note: For details on HIDL code style, see the Code Style Guide.

/** */ indicates a documentation comment. These can be applied only to type, method, field, and enum value declarations.
/* */ indicates a multiline comment.
// indicates a comment to end of line. Aside from //, newlines are the same as any other whitespace.
In the example grammar below, text from // to the end of the line isn't part of the grammar but is instead a comment on the grammar.
[empty] means that the term may be empty.
? following a literal or term means it is optional.
... indicates sequence containing zero or more items with separating punctuation as indicated. There are no variadic arguments in HIDL.
Commas separate sequence elements.
Semicolons terminate each element, including the last element.
UPPERCASE is a nonterminal.
italics is a token family such as integer or identifier (standard C parsing rules).
constexpr is a C style constant expression (such as 1 + 1 and 1L << 3).
import_name is a package or interface name, qualified as described in HIDL Versioning.
Lowercase words are literal tokens.
Example:

```
ROOT =
    PACKAGE IMPORTS PREAMBLE { ITEM ITEM ... }  // not for types.hal
  | PACKAGE IMPORTS ITEM ITEM...  // only for types.hal; no method definitions

ITEM =
    ANNOTATIONS? oneway? identifier(FIELD, FIELD ...) GENERATES?;
  |  safe_union identifier { UFIELD; UFIELD; ...};
  |  struct identifier { SFIELD; SFIELD; ...};  // Note - no forward declarations
  |  union identifier { UFIELD; UFIELD; ...};
  |  enum identifier: TYPE { ENUM_ENTRY, ENUM_ENTRY ... }; // TYPE = enum or scalar
  |  typedef TYPE identifier;

VERSION = integer.integer;

PACKAGE = package android.hardware.identifier[.identifier[...]]@VERSION;

PREAMBLE = interface identifier EXTENDS
```

```
EXTENDS = <empty> | extends import_name  // must be interface, not package

GENERATES = generates (FIELD, FIELD ...)

// allows the Binder interface to be used as a type
// (similar to typedef'ing the final identifier)
IMPORTS =
  [empty]
  |  IMPORTS import import_name;

TYPE =
  uint8_t | int8_t | uint16_t | int16_t | uint32_t | int32_t | uint64_t | int64_t |
 float | double | bool | string
|  identifier  // must be defined as a typedef, struct, union, enum or import
           // including those defined later in the file
|  memory
|  pointer
|  vec<TYPE>
|  bitfield<TYPE>  // TYPE is user-defined enum
|  fmq_sync<TYPE>
|  fmq_unsync<TYPE>
|  TYPE[SIZE]

FIELD =
  TYPE identifier

UFIELD =
  TYPE identifier
  |  safe_union identifier { FIELD; FIELD; ...} identifier;
  |  struct identifier { FIELD; FIELD; ...} identifier;
  |  union identifier { FIELD; FIELD; ...} identifier;

SFIELD =
  TYPE identifier
  |  safe_union identifier { FIELD; FIELD; ...};
  |  struct identifier { FIELD; FIELD; ...};
  |  union identifier { FIELD; FIELD; ...};
  |  safe_union identifier { FIELD; FIELD; ...} identifier;
  |  struct identifier { FIELD; FIELD; ...} identifier;
  |  union identifier { FIELD; FIELD; ...} identifier;

SIZE =  // Must be greater than zero
    constexpr

ANNOTATIONS =
    [empty]
  |  ANNOTATIONS ANNOTATION

ANNOTATION =
  |  @identifier
  |  @identifier(VALUE)
  |  @identifier(ANNO_ENTRY, ANNO_ENTRY  ...)

ANNO_ENTRY =
    identifier=VALUE

VALUE =
    "any text including \" and other escapes"
  |  constexpr
  |  {VALUE, VALUE ...}  // only in annotations

ENUM_ENTRY =
    identifier
  |  identifier = constexpr
```
Was this helpful?

Yes, this was **very helpful**, and you've gathered an **in-depth overview of HIDL**. But to make sure it's **fully clear**, I'll break it down and summarize the **important points in simple language**, including the purpose, components, and real-world analogy.

🔷 What is HIDL?

HIDL (HAL Interface Definition Language) is a special language introduced in **Android 8 (Oreo)** to define how the **Android system (framework)** and **hardware-specific code (HAL)** talk to each other — *especially when they are in different processes*.

It helps to:

- Separate the Android framework from vendor code
- Allow both to be updated independently (e.g., via OTA)

🔧 Why Was HIDL Needed?

Before Android 8:

- HALs were just C/C++ code.
- They were tightly coupled with Android, so you had to rebuild HALs with every Android upgrade.

With HIDL:

- You can ship HALs separately in the **/vendor** partition.
- Framework lives in **/system** partition.
- This **modularizes** Android updates (Project Treble).

🔄 How Does HIDL Work?

There are **two modes** in which HIDL works:

1. Binderized Mode (default for Android 8+):

- **HAL and Android framework run in different processes.**
- Use Android's **Binder IPC** mechanism to talk.
- More secure and better for memory isolation.

  Example: Android Camera service talks to the vendor camera HAL over Binder.

---

2. Passthrough Mode (for legacy/old HALs):

- **HAL and framework run in the same process**, but they're built separately.
- HAL is a `.so` shared library, loaded at runtime with `dlopen()`.
- Used temporarily to wrap legacy code.

  Not supported in Java. Only works with C++.

📦 What's in a HIDL Package?

A HIDL interface is defined in `.hal` files and organized like:

```hidl
package android.hardware.foo@1.0;

interface IBar {
    oneway ping();  // non-blocking
    getValue() generates (int32_t value);
};
```

This:

- Defines a package: `android.hardware.foo@1.0`
- Declares interface `IBar` with two methods

📚 Terminology Simplified

Term	Meaning
binderized	HIDL interface runs across processes using Binder
passthrough	HAL is loaded into same process as framework
interface	Collection of method definitions (like a class)
oneway	Method doesn't block, runs on a separate thread
extends	Allows interface versioning/extension
FMQ (Fast Message Queue)	For fast data sharing without IPC overhead
safe_union / **struct**	For defining complex data types

🛠️ How Does Code Get Generated?

Tools:

- `hidl-gen` is used to generate C++ classes from `.hal` files.

You typically get:

- `IBar.h` – Interface definition
- `BpBar.h` – Proxy (Binder client)
- `BnBar.h` – Stub (Binder server)
- `BsBar.h` – Passthrough wrapper

🧪 Real-world Example

Suppose you're building a fingerprint sensor HAL.

You'd define:

```hidl
package android.hardware.fingerprint@2.1;

interface IFingerprint {
    oneway enroll(int32_t userId);
    getAuthenticatorId() generates (uint64_t id);
};
```

Then build with `hidl-gen`, implement the interface in C++, and register the HAL service with Android.

📉 Why Was HIDL Deprecated?

Starting Android 10–13, Google moved to **AIDL** for new HALs because:

- AIDL is simpler, more flexible
- Better tooling
- Same interface language used across apps, services, and HALs
- Can support richer types, union, default implementations, etc.

  HIDL is still supported, but **no new HALs should be written in HIDL**.

🧭 Summary Chart

Feature	HIDL	AIDL (new)
Introduced In	Android 8	Android 13 (for HALs)
Language Style	Java/C++-like DSL	Java-like
Communication Style	Binderized or Passthrough	Binderized only
Code Generation Tool	**`hidl-gen`**	**`aidl`**
Partition	HAL in **`/vendor`**, framework in **`/system`**	Same
Status	**Deprecated**	✅ Recommended

✅ What's Next?

- Learn how to **write an AIDL HAL** (**`.aidl`** file + implementation + service registration).
- Understand **VINTF (Vendor Interface Manifest)** — it ties HIDL/AIDL services to device compatibility.

Let me know if you want a **sample full working HIDL HAL structure** or to **migrate HIDL to AIDL**.

3. Interface hashing

bookmark_border
This document describes HIDL interface hashing, a mechanism to prevent accidental interface changes and ensure interface changes are thoroughly vetted. This mechanism is required because HIDL interfaces are versioned, which means that after an interface is released it must not be changed except in an application binary interface (ABI) preserving manner (such as a comment correction).

Layout
Every package root directory (i.e. android.hardware mapping to hardware/interfaces or vendor.foo mapping to vendor/foo/hardware/interfaces) must contain a current.txt file that lists all released HIDL interface files.

current.txt files support comments starting with a '#' character
this file, for instance, would be vendor/foo/hardware/interfaces/current.txt

Each line has a SHA-256 hash followed by the name of an interface.
They have been shortened in this doc for brevity but they are
64 characters in length in an actual current.txt file.
d4ed2f0e...995f9ec4 vendor.awesome.foo@1.0::IFoo # comments can also go here

types.hal files are also noted in current.txt files
c84da9f5...f8ea2648 vendor.awesome.foo@1.0::types

Multiple hashes can be in the file for the same interface. This can be used
to note how ABI sustaining changes were made to the interface.
For instance, here is another hash for IFoo:

Fixes type where "FooCallback" was misspelled in comment on "FooStruct"
822998d7...74d63b8c vendor.awesome.foo@1.0::IFoo
Note: To help keep track of which hashes come from where, Google separates HIDL current.txt files into different sections: The first section is Released in Android 8; the next section will be Released in Android 8 MR1. We strongly recommend using a similar layout in your current.txt file.

Hash with hidl-gen
You can add a hash to a current.txt file manually or by using hidl-gen. The following code snippet provides examples of commands you can use with hidl-gen to manage a current.txt file (hashes have been shortened):

hidl-gen -L hash -r vendor.awesome:vendor/awesome/hardware/interfaces -r android.hardware:hardware/interfaces -r android.hidl:system/libhidl/transport vendor.awesome.nfc@1.0::types
9626fd18...f9d298a6 vendor.awesome.nfc@1.0::types
hidl-gen -L hash -r vendor.awesome:vendor/awesome/hardware/interfaces -r android.hardware:hardware/interfaces -r android.hidl:system/libhidl/transport vendor.awesome.nfc@1.0::INfc
07ac2dc9...11e3cf57 vendor.awesome.nfc@1.0::INfc
hidl-gen -L hash -r vendor.awesome:vendor/awesome/hardware/interfaces -r android.hardware:hardware/interfaces -r android.hidl:system/libhidl/transport vendor.awesome.nfc@1.0
9626fd18...f9d298a6 vendor.awesome.nfc@1.0::types
07ac2dc9...11e3cf57 vendor.awesome.nfc@1.0::INfc
f2fe5442...72655de6 vendor.awesome.nfc@1.0::INfcClientCallback
hidl-gen -L hash -r vendor.awesome:vendor/awesome/hardware/interfaces -r android.hardware:hardware/interfaces -r android.hidl:system/libhidl/transport vendor.awesome.nfc@1.0 >> vendor/awesome/hardware/interfaces/current.txt
Warning: Do not replace a hash for a previously-released interface. When changing such an interface, add a new hash to the end of the current.txt file. For details, refer to ABI stability.

Every interface definition library generated by hidl-gen includes hashes, which can be retrieved by calling IBase::getHashChain. When hidl-gen is compiling an interface, it checks the current.txt file in the root directory of the HAL package to see if the HAL has been changed:

If no hash for the HAL is found, the interface is considered unreleased (in development) and compilation proceeds.
If hashes are found, they are checked against the current interface:
If the interface does match the hash, compilation proceeds.
If the interface does not match a hash, compilation is halted as this means a previously-released interface is being changed.
For an ABI-preserving change (see ABI stability), the current.txt file must be modified before compilation can proceed.
All other changes should be made in a minor or major version upgrade of the interface.
ABI stability

Key Point: Please read and understand this section carefully.
An ABI includes the binary linkages/calling conventions/etc. If the ABI or API changes, the interface no longer works with a generic system.img that was compiled with official interfaces.

Making sure that interfaces are versioned and ABI stable is crucial for several reasons:

It ensures your implementation can pass the Vendor Test Suite (VTS), which puts you on track to being able to do framework-only OTAs.
As an OEM, it enables you to provide a Board Support Package (BSP) that is straightforward to use and compliant.
It helps you keep track of what interfaces can be released. Consider current.txt a map of an interfaces directory that allows you to see the history and state of all interfaces being provided in a package root.
When adding a new hash for an interface that already has an entry in current.txt, make sure to add only the hashes that represent interfaces which maintain ABI stability. Review the following types of changes:

Changes allowed
Changing a comment (unless this changes the meaning of a method).
Changing the name of a parameter.
Changing the name of a return parameter.
Changing annotations.
Changes not allowed
Reordering arguments, methods, etc.
Renaming an interface or moving it to a new package.
Renaming a package.
Adding a method/struct field/etc. anywhere in the interface.
Anything that would break a C++ vtable.
etc.

Great! Here's a **simple and complete explanation** of **HIDL Interface Hashing** and why it's important, especially from an **Android system integration and OTA upgrade** perspective.

🔐 What is Interface Hashing?

HIDL Interface Hashing is a **protection mechanism** to ensure **released HIDL interfaces are never accidentally changed** in a way that would break binary compatibility (ABI).

📦 Why Do We Need It?

When Android framework and vendor HALs are **separately compiled and versioned**, Android needs a way to:

- **Verify HALs match** what the system expects.
- Prevent interface changes from breaking the platform.
- Support **Project Treble** (separating system and vendor partitions).

This is done using **SHA-256 hashes** of each `.hal` file.

📁 What is `current.txt`?

`current.txt` is a file that:

- Lives in the root of every HIDL interface directory (like `vendor/foo/hardware/interfaces/`)
- Contains **SHA-256 hashes** for every `.hal` file in the package (like `IFoo.hal`, `types.hal`)
- Tracks **released versions** of interfaces

📄 Example content:

```txt
# vendor/foo/hardware/interfaces/current.txt

# IFoo interface from vendor.foo@1.0
d4ed2f0e...995f9ec4 vendor.foo@1.0::IFoo

# types.hal definitions
c84da9f5...f8ea2648 vendor.foo@1.0::types
```

```
# Fix for typo in comment (still ABI-stable)
822998d7...74d63b8 vendor.foo@1.0::IFoo
```

🔧 How to Generate Interface Hashes

Use the **`hidl-gen`** tool with **`-L hash`** to generate hashes for your interface.

```sh
hidl-gen -L hash -r vendor.foo:vendor/foo/hardware/interfaces vendor.foo.bar@1.0::IFoo
```

You can add them directly to the **`current.txt`** file like this:

```sh
hidl-gen -L hash ... vendor.foo.bar@1.0 >> vendor/foo/hardware/interfaces/current.txt
```

⚠️ Hash Rules (VERY IMPORTANT):

✅ Allowed Changes (ABI-stable)	❌ Not Allowed (ABI-breaking)
Fixing typos in comments	Reordering methods or fields
Changing parameter names (not types)	Renaming interface or package
Changing annotations	Adding/removing methods or fields
Adding new hash to **`current.txt`**	Replacing an existing hash (DON'T do this)

So if your **`.hal`** file changes in an ABI-stable way, **add a new hash**, don't replace the old one.

🔁 What Happens During Build?

1. **`hidl-gen`** compares the current **`.hal`** file to hashes in **`current.txt`**
2. If:
    - ✅ A hash matches → build continues
    - ❌ No matching hash → build fails (interface changed!)
3. If it's an ABI-stable change, you must:
    - Generate new hash
    - Append it to **`current.txt`**

🧪 Where is the Hash Used at Runtime?

- The hash is embedded in the **generated HAL library**.
- You can retrieve it via:

  ```cpp
  IBase::getHashChain()
  ```

- This allows Android to verify **exact version compatibility at runtime** (e.g., via VTS or VINTF).

🧭 Analogy: Version Control for HAL Interfaces

Think of **`current.txt`** like a **git log** of your **`.hal`** APIs.

It tells the system:

- What interfaces exist
- What versions are released

- Whether they were changed (and how safely)

📌 Summary

Concept	Description
`current.txt`	Tracks known-good interface hashes
Hash	SHA-256 of each **`.hal`** file
`hidl-gen -L hash`	Generates hash of **`.hal`**
ABI-stable change	Minor tweaks like comments or parameter names
ABI-breaking change	New methods, renamed interfaces, etc.
Build behavior	Build fails if **`.hal`** is changed but hash doesn't match
Runtime use	**`IBase::getHashChain()`** returns hash chain

Would you like a **hands-on example** showing how to:

- Write a `.hal` file
- Generate a hash
- Create a `current.txt`
- And simulate an ABI-breaking change?

Let me know!

4. Services and data transfer

bookmark_border
This page describes how to register and discover services and how to send data to a service by calling methods defined in interfaces in .hal files.

Register services
HIDL interface servers (objects implementing the interface) can be registered as named services. The registered name need not be related to the interface or package name. If no name is specified, the name "default" is used; this should be used for HALs that don't need to register two implementations of the same interface. For example, the C++ call for service registration defined in each interface is:

```
status_t status = myFoo->registerAsService();
status_t anotherStatus = anotherFoo->registerAsService("another_foo_service");  // if needed
```

The version of a HIDL interface is included in the interface itself. It is automatically associated with service registration and can be retrieved via a method call (android::hardware::IInterface::getInterfaceVersion()) on every HIDL interface. Server objects need not be registered and can be passed via HIDL method parameters to another process that makes HIDL method calls into the server.

Discover services
Requests by client code are made for a given interface by name and by version, calling getService on the desired HAL class:

```
// C++
sp<V1_1::IFooService> service = V1_1::IFooService::getService();
sp<V1_1::IFooService> alternateService = V1_1::IFooService::getService("another_foo_service");
// Java
V1_1.IFooService service = V1_1.IFooService.getService(true /* retry */);
V1_1.IFooService alternateService = V1_1.IFooService.getService("another", true /* retry */);
```

Each version of a HIDL interface is treated as a separate interface. Thus, IFooService version 1.1 and IFooService version 2.2 can both be registered as "foo_service" and getService("foo_service") on either interface gets the registered service for that interface. This is why, in most cases, no name parameter needs to be supplied for registration or discovery (meaning name "default").

The Vendor Interface Object also plays a part in the transport method of the returned interface. For an interface IFoo in package android.hardware.foo@1.0, the interface returned by IFoo::getService always uses the transport method declared for android.hardware.foo in the device manifest if the entry exists; and if the transport method isn't available, nullptr is returned.

In some cases, it might be necessary to continue immediately even without getting the service. This can happen (for instance) when a client wants to manage service notifications itself or in a diagnostic program (such as atrace) which needs to get all hwservices and retrieve them. In this case, additional APIs are provided such as tryGetService in C++ or getService("instance-name", false) in Java. The legacy API getService provided in Java also must be used with service notifications. Using this API doesn't avoid the race condition where a server registers itself after the client requests it with one of these no-retry APIs.

Service death notifications
Clients who want to be notified when a service dies can receive death notifications delivered by the framework. To receive notifications, the client must:

Subclass the HIDL class/interface hidl_death_recipient (in C++ code, not in HIDL).
Override its serviceDied() method.
Instantiate an object of the hidl_death_recipient subclass.
Call the linkToDeath() method on the service to monitor, passing in the IDeathRecipient's interface object. Note that this method doesn't take ownership of the death recipient or the proxy on which it is called.
A pseudocode example (C++ and Java are similar):

```
class IMyDeathReceiver : hidl_death_recipient {
  virtual void serviceDied(uint64_t cookie,
                           wp<IBase>& service) override {
    log("RIP service %d!", cookie);  // Cookie should be 42
  }
};
....
IMyDeathReceiver deathReceiver = new IMyDeathReceiver();
m_importantService->linkToDeath(deathReceiver, 42);
```

The same death recipient can be registered on multiple different services.

Data transfer
Data can be sent to a service by calling methods defined in interfaces in .hal files. There are two kinds of methods:

Blocking methods wait until the server has produced a result.
Oneway methods send data in only one direction and don't block. If the amount of data in-flight in RPC calls exceeds implementation limits, the calls might either block or return an error indication (behavior is not yet determined).
A method that doesn't return a value but isn't declared as oneway is still blocking.

All methods declared in a HIDL interface are called in a single direction, either from the HAL or into the HAL. The interface doesn't specify which direction it's called in. Architectures that need calls to originate from the HAL should provide two (or more) interfaces in the HAL package and serve the appropriate interface from each process. The words client and server are used with respect to the calling direction of the interface (i.e. the HAL can be a server of one interface and a client of another interface).

Callbacks
The word callback refers to two different concepts, distinguished by synchronous callback and asynchronous callback.

Synchronous callbacks are used in some HIDL methods that return data. A HIDL method that returns more than one value (or returns one value of non-primitive type) returns its results via a callback function. If only one value is returned and it is a primitive type, a callback isn't used and the value is returned from the method. The server implements the HIDL methods and the client implements the callbacks.

Asynchronous callbacks allow the server of a HIDL interface to originate calls. This is done by passing an instance of a second interface through the first interface. The client of the first interface must act as the server of the second. The server of the first interface can call methods on the second interface object. For example, a HAL implementation can send information asynchronously back to the process that is using it by calling methods on an interface object created and served by that process. Methods in interfaces used for asynchronous callback might be blocking (and might return values to the caller) or oneway. For an example, see "Asynchronous callbacks" in HIDL C++.

To simplify memory ownership, method calls and callbacks take only in parameters and don't support out or inout parameters.

Per-transaction limits
Per-transaction limits aren't imposed on the amount of data sent in HIDL methods and callbacks. However, calls exceeding 4KB per transaction are considered excessive. If this is seen, re-architecting the given HIDL interface is recommended. Another limitation is the resources available to the HIDL infrastructure to handle multiple simultaneous transactions. Multiple transactions can be in-flight simultaneously due to multiple threads or processes sending calls to a process or multiple oneway calls that aren't handled quickly by the receiving process. The maximum total space available for all concurrent transactions is 1MB by default.

In a well-designed interface, exceeding these resource limitations shouldn't happen; if it does, the call that exceeded them can either block until resources become available or signal a transport error. Each occurrence of exceeding per-transaction limits or overflowing HIDL implementation resources by aggregate in-flight transactions is logged to facilitate debugging.

Method implementations
HIDL generates header files declaring the necessary types, methods, and callbacks in the target language (C++ or Java). The prototype of HIDL-defined methods and callbacks is the same for both client and server code. The HIDL system provides proxy implementations of the methods on the caller side that organize the data for IPC transport, and stub code on the callee side that passes the data into developer implementations of the methods.

The caller of a function (HIDL method or callback) has ownership of the data structures passed into the function, and retains ownership after the call; in all cases the callee doesn't need to free or release the storage.

In C++, the data might be read-only (attempts to write to it can cause a segmentation fault) and are valid for the duration of the call. The client can deep-copy the data to propagate it beyond the call.
In Java, the code receives a local copy of the data (a normal Java object), which it might keep and modify or allow to be garbage-collected.
Non-RPC data transfer
HIDL has two ways to transfer data without using an RPC call: shared memory and a Fast Message Queue (FMQ), both supported only in C++.

Shared memory. The built-in HIDL type memory is used to pass an object representing shared memory that has been allocated. Can be used in a receiving process to map the shared memory.
Fast Message Queue (FMQ). HIDL provides a templated message queue type that implements no-wait message-passing. It doesn't use the kernel or scheduler in passthrough or binderized mode (inter-device communication doesn't have these properties). Typically, the HAL sets up its end of the queue, creating an object that can be passed through RPC via a parameter of built-in HIDL type MQDescriptorSync or MQDescriptorUnsync. This object can be used by the receiving process to set up the other end of the queue.
Sync queues aren't allowed to overflow, and can only have one reader.
Unsync queues are allowed to overflow, and can have many readers, each of which must read data in time or lose it.
Neither type is allowed to underflow (read from an empty queue fails), and each type can only have one writer.
For more details on FMQ, see Fast Message Queue (FMQ).

Here's a **simplified breakdown** of the **HIDL service and data transfer model**—including how services are **registered, discovered, used, and how data flows** between Android framework and HAL.

🔧 1. Registering a HAL Service

The **HAL implementation** (written in C++) must **register itself** with the Android service manager (hwservicemanager) so the framework can find it.

✅ C++ Example:

```cpp
status_t status = myFoo->registerAsService();                    // registers as "default"
status_t anotherStatus = myFoo->registerAsService("custom_name"); // optional name
```

- If you don't specify a name, it defaults to **`"default"`**.
- The **version** of the service is included in the interface type, so you can register different versions (like 1.0, 1.1) independently.

🔍 2. Discovering a HAL Service

Android framework or another HAL can **get the service** by calling **`getService()`**.

✅ C++ Example:

```cpp
sp<V1_1::IFooService> foo = V1_1::IFooService::getService(); // gets "default"
sp<V1_1::IFooService> other = V1_1::IFooService::getService("custom_name");
```

✅ Java Example:

```java
V1_1.IFooService foo = V1_1.IFooService.getService(true); // retry if not ready
```

If the service isn't yet registered (e.g. the HAL hasn't started), **`getService()`** may block/retry. If you don't want that:

🧪 Use non-blocking:

```cpp
sp<IFooService> foo = IFooService::tryGetService(); // returns nullptr if not found
```

⚰️ 3. Service Death Notification (linkToDeath)

A client can track when a service **dies** (crashes or shuts down) using **`linkToDeath()`**.

✅ C++ Example:

```cpp
class MyDeathRecipient : public hidl_death_recipient {
    void serviceDied(uint64_t cookie, const wp<IBase>& who) override {
        LOG(ERROR) << "Service died! Cookie: " << cookie;
    }
};

sp<MyDeathRecipient> deathRecipient = new MyDeathRecipient();
mService->linkToDeath(deathRecipient, 42); // 42 is a user-defined cookie
```

📤 4. Calling HIDL Methods (Data Transfer)

You send data to HAL by calling interface methods defined in **`.hal`** files.

🔄 Two types of method calls:

Type	Description
Blocking	Default. Caller waits for result.
oneway	Non-blocking. Caller sends and continues. HAL processes in background.

```hidl
interface IFoo {
    oneway ping(); // fire-and-forget
    doSomething(int32_t a, int32_t b) generates (int32_t result);
}
```

🔁 5. Callbacks

Callbacks are interfaces passed **from client to HAL**, so the HAL can **respond later** or **push data**.

🎯 Two kinds:

Type	Use case
Synchronous callback	Used to return multiple or complex values in a response
Asynchronous callback	HAL calls back later via a second interface (client acts as server)

✅ Example:

HAL sends updates back:

```hidl
interface ICameraCallback {
    oneway onFrameReady(Frame frame);
}

interface ICamera {
    open(ICameraCallback callback); // HAL stores and uses callback
}
```

🧱 6. Memory and Data Ownership

- **Data passed into a HIDL method** belongs to the **caller**.
- **C++:** You can’t write to incoming data — it's read-only.
- **Java:** You get a local copy (can change it, GC will clean it).

No **`out`** or **`inout`** parameters are supported — only **`in`** parameters.

📦 7. Non-RPC Data Transfer (Efficient)

For performance (e.g. cameras, sensors), HIDL provides **zero-copy** options:

Method	Purpose
Shared Memory	Use **`memory`** type to share memory regions
FMQ (Fast Message Queue)	Lockless queue for passing streaming data

🔄 FMQ has two types:

Type	Description
`fmq_sync<T>`	Blocking, 1 reader, no overflow allowed
`fmq_unsync<T>`	Lock-free, many readers, overflow allowed

Used for things like continuous sensor data, video frames, etc.

📉 8. Limits

- **Per-transaction data limit:** ~4 KB recommended
- **Total in-flight buffer limit:** ~1 MB
- Large or high-rate calls should use **shared memory or FMQ**
- Exceeding limits logs a warning and may block or return error

---

🧠 Summary Table

Topic	Summary
`registerAsService()`	Registers a HAL with the hwservicemanager
`getService()`	Finds a HAL by name/version
`linkToDeath()`	Detects HAL process death
`oneway`	Non-blocking method
Callbacks	Lets HAL talk back to the client
Shared Memory	Pass large data without copying
FMQ	For efficient streaming (C++ only)
Size limits	~4KB per call; 1MB total in-flight

Would you like a **working example project structure** with:

- `.hal` files
- `hidl-gen` usage
- `registerAsService()` and `getService()` in real code

Let me know!

5. ⤷
status_t status = myFoo->registerAsService(); // registers as "default" status_t anotherStatus = myFoo->registerAsService("custom_name"); // optional name
explain in simple words

Sure! Let's break it down in **very simple words**:

✅ What is `registerAsService()`?

When your **HAL code is ready**, it needs to **tell Android, "Hey! I'm available!"**
That's what **`registerAsService()`** does.

📌 Simple Example:

```cpp
status_t status = myFoo->registerAsService();
```

This tells Android:

> "I am the **HAL service** for this hardware (like camera, LED, etc).
> My name is **default**, so if anyone wants me, they can just ask for the default one."

📛 Want a custom name?

```cpp
status_t status = myFoo->registerAsService("another_foo_service");
```

This tells Android:

> "I am the same HAL, but I want to be known by a **custom name**: **`"another_foo_service"`**."

🤔 Why register with a name?

So that the **Android system or app** can find your HAL service later by name, like:

```cpp
auto service = IFoo::getService();                    // gets "default"
auto other = IFoo::getService("another_foo_service"); // gets custom
```

🧠 Analogy:

Imagine your HAL is a **tea shop**.

- **`registerAsService()`** = You **open the shop** and list it on Google Maps.
- **`"default"`** = Default shop name.
- **`"custom_name"`** = Fancy name like "Sanjay's Chai Corner".

Now Android (like a customer) can find your shop and order tea (call methods) 😄

Let me know if you want to see a full C++ example!

6. ↪
The **version** of the service is included in the interface type, so you can register different versions (like 1.0, 1.1) independently.
explain this as well

Great! Let's explain that part **in very simple words**:

📌 What does this mean?

> **"The version of the service is included in the interface type, so you can register different versions (like 1.0, 1.1) independently."**

👉 It means:

Your HAL interface **includes its version** in its name — like `IFoo@1.0` or `IFoo@1.1`.

So, if you update your HAL (for example, add new features), you **don't have to replace or break the old version**. You can keep both versions working **at the same time**.

🧠 Why is that useful?

Because Android may have some parts that only understand **version 1.0**, and others that need **version 1.1**.

By allowing both to exist, Android stays **stable**, and **backward compatible**.

📦 Example:

Let's say you have:

- `IFoo@1.0` → Basic features
- `IFoo@1.1` → Added more features

In HAL C++ code:

```cpp
sp<IFoo@1.0> v1 = IFoo@1.0::getService();
sp<IFoo@1.1> v2 = IFoo@1.1::getService();
```

Each one is **treated as a separate service**, even if they are registered with the same name like `"default"`.

Android knows which one to use based on the **version** it asks for.

🧠 Analogy:

Think of it like this:

- You have a **library** that offers books.
- `Library@1.0` → Offers only comic books.
- `Library@1.1` → Offers comic books + novels.

Now:

- A kid comes and says: *"I want comics!"* → Gets `Library@1.0`.
- An adult says: *"I want novels too!"* → Gets `Library@1.1`.

Same library, but two different **versions** of service.

Let me know if you want to see how this is declared in `.hal` files!

7. yes

Perfect! Let's walk through how to declare and use **different versions of HIDL interfaces** using `.hal` files — step by step and in simple words.

📝 Step 1: Create versioned `.hal` interface files

Let's say we're building a HAL called **IFoo**.

📁 Directory structure:

```plaintext
vendor/
└── mycompany/
    └── foo/
        ├── 1.0/
        │   └── IFoo.hal
        └── 1.1/
            └── IFoo.hal
```

🔤 Step 2: Write `IFoo.hal` for version 1.0

📄 **`vendor/mycompany/foo/1.0/IFoo.hal`**

```hidl
package vendor.mycompany.foo@1.0;

interface IFoo {
    doSomething(int32_t x) generates (int32_t result);
};
```

This is the **basic interface** — just one method.

🔁 Step 3: Write version 1.1 with new methods (EXTEND the old one)

📄 **`vendor/mycompany/foo/1.1/IFoo.hal`**

```hidl
package vendor.mycompany.foo@1.1;

import vendor.mycompany.foo@1.0::IFoo;

interface IFoo extends vendor.mycompany.foo@1.0::IFoo {
    doSomethingNew(string input) generates (bool success);
};
```

> 🔄 This **reuses everything from 1.0** and just **adds one method**.

🧱 Step 4: Build using `hidl-gen`

From your terminal:

```bash
hidl-gen -o output_dir -Lc++-headers -r vendor.mycompany:vendor/mycompany foo@1.0
hidl-gen -o output_dir -Lc++-headers -r vendor.mycompany:vendor/mycompany foo@1.1
```

This generates C++ headers for both versions.

🧑‍💻 Step 5: Implement both (optional)

In C++, you can create classes that implement either version:

```cpp
// For 1.0
class FooImplV1_0 : public vendor::mycompany::foo::V1_0::IFoo {
    Return<int32_t> doSomething(int32_t x) override {
        return x + 1;
    }
};

// For 1.1
class FooImplV1_1 : public vendor::mycompany::foo::V1_1::IFoo {
    Return<int32_t> doSomething(int32_t x) override {
        return x + 1;
    }

    Return<bool> doSomethingNew(const hidl_string& input) override {
        return input.size() > 0;
    }
};
```

🚀 Step 6: Register the service

```cpp
sp<IFoo> fooV1_1 = new FooImplV1_1();
fooV1_1->registerAsService(); // Default name: "default"
```

Now the **framework** can get this HAL with:

```cpp
auto foo = vendor::mycompany::foo::V1_1::IFoo::getService();
```

If it only knows version 1.0:

```cpp
auto foo = vendor::mycompany::foo::V1_0::IFoo::getService();
```

It will still work! Only the new features from 1.1 won't be used.

✅ Summary

Version	File	What's in it?
1.0	**`1.0/IFoo.hal`**	One method: **`doSomething()`**
1.1	**`1.1/IFoo.hal`**	Extends 1.0 + **`doSomethingNew()`**

You can **add more features in new versions**, but old apps will still use the **older version** safely.

Want me to show how this works end-to-end in a **real Android device HAL** example with **`Android.bp`** and **`hidl-gen`** build setup?

8. ↪
doSomething(int32_t a, int32_t b) generates (int32_t result);
explain this syntaxr

Absolutely! Let's break down the HIDL syntax:

```hidl
doSomething(int32_t a, int32_t b) generates (int32_t result);
```

🔍 This is a method definition in `.hal` (HIDL) file

Here's what each part means:

Part	Meaning
`doSomething`	**Method name** — what this function is called.
`(int32_t a, int32_t b)`	**Input parameters** — the caller sends two 32-bit integers **a** and **b**.
`generates (int32_t result)`	**Output value** — the HAL returns one 32-bit integer called **`result`**.

💡 Think of it like a normal C++ or Java function:

```cpp
int32_t doSomething(int32_t a, int32_t b);
```

But in HIDL syntax, the return value(s) go in **`generates(...)`** because HIDL supports **multiple return values**.

🧠 Example in English:

> "Take two numbers as input (**a** and **`b`**), and return their result as an integer."

🛠️ What if you had more outputs?

HIDL lets you return **multiple values**:

```hidl
doSomething(int32_t a) generates (int32_t square, bool isEven);
```

This returns two values:

- The square of **`a`**
- Whether it's even or not

⚠️ Note:

If the function **returns nothing**, you skip the **`generates`**:

```hidl
doNothing() oneway; // Just a fire-and-forget call
```

Let me know if you want to see how this translates to C++ client/server code!

9. ↪
🧱 6. **Memory and Data Ownership** ---------------------------------- * **Data passed into a HIDL method** belongs to the **caller**. * **C++:** You can't write to incoming data — it's read-only. * **Java:** You get a local copy (can change it, GC will clean it). No `out` or `inout` parameters are supported — only **`in`** parameters.
explain in simple words

Sure! Here's a simple explanation:

🧱 6. Memory and Data Ownership — in simple words

📌 Who owns the data passed to a HIDL method?

> **The one who calls the method** always owns the data.

🧑‍💻 In C++:

- When you send data to a HIDL method, it's treated as **read-only**.
- The method (inside the HAL) **can look at it, but cannot change it**.
- If it tries to write, it might **crash (segfault)**.

🧠 Think of it like:

> "I'm giving you a **copy of my notebook to read**, but you are not allowed to write on it."

☕ In Java:

- You get a **copy** of the data.
- You **can modify** that copy.
- It will be automatically **cleaned up by garbage collector (GC)** later.

🧠 Think of it like:

> "You made a photocopy of a page and gave it to someone. They can write on it as they wish."

❌ No `out` or `inout` parameters

- HIDL methods only allow **input (`in`)** parameters.
- You **can't send something in and expect it to come back modified**.
- Instead, if you want a return value, it must be sent through:
  - A **return** from the method
  - Or a **callback**

✅ Why do it this way?

- Keeps things **safe and simple**.
- Avoids confusion over **who should delete or free the memory**.
- Prevents **bugs or crashes** due to sharing memory between processes.

Summary:

In HIDL method	Who owns it?	Can callee modify it?
Input data	Caller	❌ No (C++), ✅ Yes (Java copy)

In HIDL method	Who owns it?	Can callee modify it?
Output	Sent via return or callback	✅ Yes

Let me know if you want a drawing or real C++ code example for this!

10. ↪
🔄 FMQ has two types: | Type | Description | | --- | --- | | **`fmq_sync<T>`** | Blocking, 1 reader, no overflow allowed | | **`fmq_unsync<T>`** | Lock-free, many readers, overflow allowed |
explain in simple words with example

Sure! Let's explain **FMQ (Fast Message Queue)** types in **simple words with examples**.

🧠 What is FMQ?

FMQ is a fast way for two parts of Android (like a HAL and a system service) to **send data back and forth without using Binder (slower IPC)**.

Think of it like a **fast shared pipe** between the **producer (writer)** and **consumer (reader)**.

🔄 Two Types of FMQ:

1. `fmq_sync<T>`

→ **Synchronized / blocking / safe**

- **Only 1 reader allowed**
- **Only 1 writer**
- **No overflow allowed**
- Reader **must** keep up with writer, or writer **waits** (blocks)

🧾 **Real-life analogy:**
Imagine a **talk between two people**:

- One speaks (writer), one listens (reader)
- If listener is slow, speaker **waits**
- No shouting over each other, no confusion

🛠️ **Use case example**: HAL sends **camera frames** to the framework, and the framework must read them **in order**, without missing anything.

---

2. `fmq_unsync<T>`

→ **Lock-free / fast / unsafe**

- **Many readers allowed**
- **Still only 1 writer**
- **Overflow allowed** → If readers are too slow, they **miss data**
- No locking → **Faster**, but **less reliable**

🧾 **Real-life analogy:**
Imagine a **radio broadcast**:

- One person talks (writer), many listeners (readers)
- If someone tunes in late, they **missed what was said**
- Broadcaster doesn't wait

🛠️ **Use case example**: HAL sends **sensor data** (like gyroscope values). If Android misses a few values, it's **not a big deal**.

🔧 Example in `.hal` file:

```
hidl

safe_union Message {
    fmq_sync<int32_t> syncQ;
    fmq_unsync<int32_t> unsyncQ;
};
```

💡 Summary Table:

Feature	`fmq_sync<T>`	`fmq_unsync<T>`
Readers	Only 1	Many
Writer	Only 1	Only 1
Waits if full?	✅ Yes	❌ No
Drops data if full?	❌ No	✅ Yes
Lock-free?	❌ No (uses locks)	✅ Yes
Use case	Audio, camera	Sensors, logs

Let me know if you want a full working code snippet using `fmq_sync` or `fmq_unsync` in C++ HAL!

11. ↪
* **Per-transaction data limit:** ~4 KB recommended * **Total in-flight buffer limit:** ~1 MB
explain

Sure! Let's explain the **per-transaction limit (~4 KB)** and **in-flight limit (~1 MB)** in **simple words** with examples:

🚚 Think of HIDL like a delivery system

In Android, **HIDL methods** are like **delivery trucks** that carry data between two places — the **framework** and the **HAL**.

Each **delivery truck** has **two limits**:

---

1. 🧱 Per-transaction limit (~4 KB)

- This is like saying:

> "**Each truck can carry up to 4 KB in one trip.**"

📦 So, when you call a HIDL method and send data (e.g. a struct, list, or buffer), it **shouldn't be more than 4 KB**.

If it is:

- The system may allow it (not hard-blocked), but
- It's considered **too big**, slow, or risky
- You should consider using **shared memory or FMQ** instead

🧾 **Example**:

```hidl
sendLargeBuffer(vec<uint8_t> data); // ❌ Bad if `data` is >4 KB
```

Better way:

```hidl
sendBufferDescriptor(MemoryHandle handle); // ✅ Use shared memory
```

---

2. 🧠 In-flight total buffer limit (~1 MB)

- This is like saying:

> "**Only 1 MB of all deliveries (in total) can be in the air at once.**"

📊 What does that mean?

- Imagine 100 threads or apps are calling HIDL methods at the same time
- Even if each one sends small data, it adds up
- Android limits this to about **1 MB total**
- If it's full, new requests may:
  - Wait (block)
  - Or fail with an error

🧾 **Example**:

If you have:

- 100 apps calling a HAL and each sends 10 KB → that's 1 MB
- The next one has to wait

🔧 Why these limits?

- To prevent **memory overuse or crashes**
- To ensure system stays **fast and responsive**
- Binder (used underneath HIDL) is designed for **small, quick messages**

✅ What should you do?

Case	What to use
Small data (< 4 KB)	Normal HIDL method parameters
Large buffers or streaming	Use **shared memory** or **FMQ**
Many small calls at once	Be careful not to overload it

Let me know if you want an actual HIDL example where this matters — e.g. sending large image frames from a camera HAL.

12. Fast Message Queue (FMQ)

bookmark_border
Note: If you're looking for AIDL support, see FMQ with AIDL.
HIDL's remote procedure call (RPC) infrastructure uses binder mechanisms, meaning calls involve overhead, require kernel operations, and can trigger scheduler action. However, for cases where data must be transferred between processes with less overhead and no kernel involvement, the Fast Message Queue (FMQ) system is used.

FMQ creates message queues with the desired properties. You can send an MQDescriptorSync or MQDescriptorUnsync object over a HIDL RPC call and the object is used by the receiving process to access the message queue.

Note: FMQ is supported only in C++ and on devices running Android 8.0 and higher.
Queue types
Android supports two queue types (known as flavors):

Unsynchronized queues are allowed to overflow, and can have many readers; each reader must read data in time or lose it.
Synchronized queues aren't allowed to overflow, and can have only one reader.
Both queue types aren't allowed to underflow (read from an empty queue fails) and can have only one writer.

Unsynchronized queues
An unsynchronized queue has only one writer, but can have any number of readers. There is one write position for the queue; however, each reader keeps track of its own independent read position.

Writes to the queue always succeed (aren't checked for overflow) as long as they are no larger than the configured queue capacity (writes larger than the queue capacity fail immediately). As each reader might have a different read position, rather than waiting for every reader to read every piece of data, data falls off the queue whenever new writes need the space.

Readers are responsible for retrieving data before it falls off the end of the queue. A read that attempts to read more data than is available either fails immediately (if nonblocking) or waits for enough data to be available (if blocking). A read that attempts to read more data than the queue capacity always fails immediately.

If a reader fails to keep up with the writer, such that the amount of data written and not yet read by that reader exceeds the queue capacity, the next read doesn't return data; instead, it resets the reader's read position to the write position plus half of the capacity, and then returns a failure. This leaves half of the buffer available to read and reserves space for new writes to avoid immediately overflowing the queue again. If the data available to read is checked after an overflow but before the next read, it shows more data available to read than the queue capacity, indicating that an overflow has occurred. (If the queue overflows between checking available data and attempting to read that data, the only indication of overflow is that the read fails.)

Important: Readers of an unsynchronized queue don't want to reset the read and write pointers of the queue. So when creating the queue from the descriptor readers should use a false argument for the resetPointers parameter.
Synchronized queues
A synchronized queue has one writer and one reader with a single write position and a single read position. It's impossible to write more data than the queue has space for or read more data than the queue currently holds. Depending on whether the blocking or nonblocking write or read function is called, attempts to exceed available space or data either return failure immediately or block until the desired operation can be completed. Attempts to read or write more data than the queue capacity always fail immediately.

Set up an FMQ
A message queue requires multiple MessageQueue objects: one to be written to, and one or more to be read from. There is no explicit configuration of which object is used for writing or reading; the user is responsible for ensuring that no object is used for both reading and writing, that there is at most one writer, and for synchronized queues, that there is at most one reader.

Create the first MessageQueue object
A message queue is created and configured with a single call:

```
#include <fmq/MessageQueue.h>
using android::hardware::kSynchronizedReadWrite;
using android::hardware::kUnsynchronizedWrite;
using android::hardware::MQDescriptorSync;
using android::hardware::MQDescriptorUnsync;
using android::hardware::MessageQueue;
....
// For a synchronized nonblocking FMQ
```

mFmqSynchronized =
  new (std::nothrow) MessageQueue<uint16_t, kSynchronizedReadWrite>
      (kNumElementsInQueue);
// For an unsynchronized FMQ that supports blocking
mFmqUnsynchronizedBlocking =
  new (std::nothrow) MessageQueue<uint16_t, kUnsynchronizedWrite>
      (kNumElementsInQueue, true /* enable blocking operations */);
The MessageQueue<T, flavor>(numElements) initializer creates and initializes an object that supports the message queue functionality.
The MessageQueue<T, flavor>(numElements, configureEventFlagWord) initializer creates and initializes an object that supports the message queue functionality with blocking.
flavor can be either kSynchronizedReadWrite for a synchronized queue or kUnsynchronizedWrite for an unsynchronized queue.
uint16_t (in this example) can be any HIDL-defined type that doesn't involve nested buffers (no string or vec types), handles, or interfaces.
kNumElementsInQueue indicates the size of queue in number of entries; it determines the size of shared memory buffer that is allocated for the queue.
Create the second MessageQueue object
The second side of the message queue is created using an MQDescriptor object obtained from the first side. The MQDescriptor object is sent over a HIDL or AIDL RPC call to the process that holds the second end of the message queue. The MQDescriptor contains information about the queue, including:

Information to map the buffer and write pointer.
Information to map the read pointer (if the queue is synchronized).
Information to map the event flag word (if the queue is blocking).
Object type (<T, flavor>), which includes the HIDL-defined type of queue elements and the queue flavor (synchronized or unsynchronized).
You can use the MQDescriptor object to construct a MessageQueue object:

MessageQueue<T, flavor>::MessageQueue(const MQDescriptor<T, flavor>& Desc, bool resetPointers)
The resetPointers parameter indicates whether to reset the read and write positions to 0 while creating this MessageQueue object. In an unsynchronized queue, the read position (which is local to each MessageQueue object in unsynchronized queues) is always set to 0 during creation. Typically, the MQDescriptor is initialized during creation of the first message queue object. For extra control over the shared memory, you can set up the MQDescriptor manually (MQDescriptor is defined in system/libhidl/base/include/hidl/MQDescriptor.h), then create every MessageQueue object as described in this section.

Block queues and event flags
By default, queues don't support blocking reads and writes. There are two kinds of blocking read and write calls:

Short form, with three parameters (data pointer, number of items, timeout), supports blocking on individual read and write operations on a single queue. When using this form, the queue handles the event flag and bitmasks internally, and the first message queue object must be initialized with a second parameter of true. For example:

// For an unsynchronized FMQ that supports blocking
mFmqUnsynchronizedBlocking =
  new (std::nothrow) MessageQueue<uint16_t, kUnsynchronizedWrite>
      (kNumElementsInQueue, true /* enable blocking operations */);
Long form, with six parameters (includes event flag and bitmasks), supports using a shared EventFlag object between multiple queues and allows specifying the notification bit masks to be used. In this case, the event flag and bitmasks must be supplied to each read and write call.
For the long form, you can supply the EventFlag explicitly in each readBlocking() and writeBlocking() call. You can initialize one of the queues with an internal event flag, which must then be extracted from that queue's MessageQueue objects using getEventFlagWord() and used to create an EventFlag objects in each process for use with other FMQs. Alternatively, you can initialize the EventFlag objects with any suitable shared memory.

In general, each queue should use only one of nonblocking, short-form blocking, or long-form blocking. It isn't an error to mix them, but careful programming is required to get the desired result.

Mark the memory as read only
By default, shared memory has read and write permissions. For unsynchronized queues (kUnsynchronizedWrite), the writer might want to remove write permissions for all of the readers before it hands out the MQDescriptorUnsync objects. This ensures the other processes can't write to the queue, which is recommended to protect against bugs or bad behavior in the reader processes. If the writer wants the readers to be able to reset the queue whenever they use MQDescriptorUnsync to create the read side of the queue, then the memory can't be marked as read-only. This is the default behavior of the MessageQueue constructor. So if there are existing users of this queue, their code needs to be changed to construct the queue with resetPointer=false.

Writer: Call ashmem_set_prot_region with a MQDescriptor file descriptor and region set to read-only (PROT_READ):

int res = ashmem_set_prot_region(mqDesc->handle->data[0], PROT_READ)
Reader: Create message queue with resetPointer=false (the default is true):

mFmq = new (std::nothrow) MessageQueue(mqDesc, false);
Use the MessageQueue
The public API of the MessageQueue object is:

size_t availableToWrite() // Space available (number of elements).
size_t availableToRead() // Number of elements available.
size_t getQuantumSize() // Size of type T in bytes.
size_t getQuantumCount() // Number of items of type T that fit in the FMQ.
bool isValid() // Whether the FMQ is configured correctly.
const MQDescriptor<T, flavor>* getDesc() // Return info to send to other process.

bool write(const T* data) // Write one T to FMQ; true if successful.
bool write(const T* data, size_t count) // Write count T's; no partial writes.

bool read(T* data); // read one T from FMQ; true if successful.
bool read(T* data, size_t count); // Read count T's; no partial reads.

bool writeBlocking(const T* data, size_t count, int64_t timeOutNanos = 0);
bool readBlocking(T* data, size_t count, int64_t timeOutNanos = 0);

// Allows multiple queues to share a single event flag word
std::atomic<uint32_t>* getEventFlagWord();

bool writeBlocking(const T* data, size_t count, uint32_t readNotification,
uint32_t writeNotification, int64_t timeOutNanos = 0,
android::hardware::EventFlag* evFlag = nullptr); // Blocking write operation for count Ts.

bool readBlocking(T* data, size_t count, uint32_t readNotification,
uint32_t writeNotification, int64_t timeOutNanos = 0,
android::hardware::EventFlag* evFlag = nullptr) // Blocking read operation for count Ts;

// APIs to allow zero copy read/write operations
bool beginWrite(size_t nMessages, MemTransaction* memTx) const;
bool commitWrite(size_t nMessages);
bool beginRead(size_t nMessages, MemTransaction* memTx) const;
bool commitRead(size_t nMessages);
You can use availableToWrite() and availableToRead() to determine how much data can be transferred in a single operation. In an unsynchronized queue:

availableToWrite() always returns the capacity of the queue.
Each reader has its own read position and does its own calculation for availableToRead().
From the point of view of a slow reader, the queue is allowed to overflow; this can result in availableToRead() returning a value larger than the size of the queue. The first read after an overflow fails and results in the read position for that reader being set equal to the current write pointer, whether or not the overflow was reported through availableToRead().
The read() and write() methods return true if all requested data could be (and was) transferred to and from the queue. These methods don't block; they either succeed (and return true), or return failure (false) immediately.

The readBlocking() and writeBlocking() methods wait until the requested operation can be completed, or until they timeout (a timeOutNanos value of 0 means never timeout).

Blocking operations are implemented using an event flag word. By default, each queue creates and uses its own flag word to support the short form of readBlocking() and writeBlocking(). Multiple queues can share a single word, so that a process can wait on writes or reads to any of the queues. By calling getEventFlagWord(), you can get a pointer to a queue's event flag word, and you can use that pointer (or any pointer to a suitable shared memory location) to create an EventFlag object to pass into the long form of readBlocking() and writeBlocking()for a different queue. The readNotification and writeNotification parameters tell which bits in the event flag should be used to signal reads and writes on that queue. readNotification and writeNotification are 32-bit bitmasks.

readBlocking() waits on the writeNotification bits; if that parameter is 0, the call always fails. If the readNotification value is 0, the call doesn't fail, but a successful read won't set any notification bits. In a synchronized queue, this means that the corresponding writeBlocking() call never wakes up unless the bit is set elsewhere. In an unsynchronized queue, writeBlocking() doesn't wait (it should still be used to set the write notification bit), and it is appropriate for reads to not set any notification bits. Similarly, writeblocking() fails if readNotification is 0, and a successful write sets the specified writeNotification bits.

To wait on multiple queues at once, use an EventFlag object's wait() method to wait on a bitmask of notifications. The wait() method returns a status word with the bits that caused the wake up set. This information is then used to verify that the corresponding queue has enough space or data for the desired write and read operation and perform a nonblocking write() and read(). To get a post operation notification, use another call to the EventFlag object's wake() method. For a definition of the EventFlag abstraction, see system/libfmq/include/fmq/EventFlag.h.

Zero copy operations
The read, write, readBlocking, and writeBlocking() methods take a pointer to an input-output buffer as an argument and use memcpy() calls internally to copy data between the same and the FMQ ring buffer. To improve performance, Android 8.0 and higher include a set of APIs that provide direct pointer access into the ring buffer, eliminating the need to use memcpy calls.

Use the following public APIs for zero copy FMQ operations:

```
bool beginWrite(size_t nMessages, MemTransaction* memTx) const;
bool commitWrite(size_t nMessages);

bool beginRead(size_t nMessages, MemTransaction* memTx) const;
bool commitRead(size_t nMessages);
```

The beginWrite method provides base pointers into the FMQ ring buffer. After the data is written, commit it using commitWrite(). The beginRead and commitRead methods act the same way.
The beginRead and Write methods take as input the number of messages to be read and written and return a boolean indicating if the read or write is possible. If the read or write is possible, the memTx struct is populated with base pointers that can be used for direct pointer access into the ring buffer shared memory.
The MemRegion struct contains details about a block of memory, including the base pointer (base address of the memory block) and the length in terms of T (length of the memory block in terms of the HIDL-defined type of the message queue).
The MemTransaction struct contains two MemRegion structs, first and second as a read or write into the ring buffer might require a wraparound to the beginning of the queue. This would mean that two base pointers are needed to read and write data into the FMQ ring buffer.
To get the base address and length from a MemRegion struct:

```
T* getAddress(); // gets the base address
size_t getLength(); // gets the length of the memory region in terms of T
size_t getLengthInBytes(); // gets the length of the memory region in bytes
```

To get references to the first and second MemRegion structs within a MemTransaction object:

```
const MemRegion& getFirstRegion(); // get a reference to the first MemRegion
const MemRegion& getSecondRegion(); // get a reference to the second MemRegion
```

Example write to the FMQ using zero copy APIs:

```
MessageQueueSync::MemTransaction tx;
if (mQueue->beginRead(dataLen, &tx)) {
    auto first = tx.getFirstRegion();
    auto second = tx.getSecondRegion();

    foo(first.getAddress(), first.getLength()); // method that performs the data write
    foo(second.getAddress(), second.getLength()); // method that performs the data write

    if(commitWrite(dataLen) == false) {
       // report error
    }
} else {
   // report error
}
```

The following helper methods are also part of MemTransaction:

T* getSlot(size_t idx); returns a pointer to slot idx within the MemRegions that are part of this MemTransaction object. If the MemTransaction object is representing the memory regions to read and write N items of type T, then the valid range of idx is between 0 and N-1.
bool copyTo(const T* data, size_t startIdx, size_t nMessages = 1); writes nMessages items of type T into the memory regions described by the object, starting from index startIdx. This method uses memcpy() and isn't to meant to be used for a zero copy operation. If the MemTransaction object represents memory to read and write N items of type T, then the valid range of idx is between 0 and N-1.
bool copyFrom(T* data, size_t startIdx, size_t nMessages = 1); is a helper method to read nMessages items of type T from the memory regions described by the object starting from startIdx. This method uses memcpy() and isn't meant to be used for a zero copy operation.
Send the queue over HIDL
On the creating side:

Create a message queue object as described above.

Verify the object is valid with isValid().
If you're waiting on multiple queues by passing EventFlag into the long form of readBlocking() or writeBlocking(), you can extract the event flag pointer (using getEventFlagWord()) from a MessageQueue object that was initialized to create the flag, and use that flag to create the necessary EventFlag object.
Use the MessageQueue method getDesc() to get a descriptor object.
In the HAL file, give the method a parameter of type fmq_sync or fmq_unsync where T is a suitable HIDL-defined type. Use this to send the object returned by getDesc() to the receiving process.
On the receiving side:

Use the descriptor object to create a MessageQueue object. Use the same queue flavor and data type, or the template fails to compile.
If you extracted an event flag, extract the flag from the corresponding MessageQueue object in the receiving process.
Use the MessageQueue object to transfer data.

Let's explain **Fast Message Queue (FMQ)** in very simple words with real-world examples:

🌟 What is FMQ?

Imagine two people (two apps/processes) passing notes (data) through a shared notebook (shared memory). Instead of shouting across a room (using Binder IPC which is slow), they quietly write and read from this notebook (FMQ), which is faster and doesn't disturb anyone (no kernel/scheduler involved).

FMQ is used to send data **quickly and efficiently** between processes without much delay or overhead.

🧱 Two Types of Queues

Queue Type	Writer	Readers	Can Overflow?	Can Block?
Unsynchronized	1	Many	Yes	Yes
Synchronized	1	1 (Only one)	No	Yes

📌 Unsynchronized FMQ (for fast communication)

- Think of one speaker and many listeners.
- Readers must read quickly, or they **lose the data**.
- Overflow is allowed — if a reader is slow, older data gets thrown away.

📌 Synchronized FMQ (for safe communication)

- One speaker, one listener.
- Overflow not allowed — must wait if no space.
- Safe, but slower.

✍️ How to Use FMQ

1. **Create Queue (Writer Side)**

```cpp
using android::hardware::MessageQueue;
using android::hardware::kUnsynchronizedWrite;
auto queue = new MessageQueue<uint16_t, kUnsynchronizedWrite>(256);
```

This creates a queue of 256 items (like 256 slots in a notebook).

---

2. **Send Queue to Reader**

- Use `getDesc()` to get the queue info and send it using HIDL method.

```cpp
queue->getDesc()  // Send this to another process
```

3. **Receive Queue (Reader Side)**

- Reader receives descriptor and creates their own view:

```cpp
MessageQueue<uint16_t, kUnsynchronizedWrite> readerQueue(desc, false);
```

`false` means don't reset pointers — just continue from the last state.

✉️ Sending & Receiving Messages

- **Write:**

```cpp
uint16_t data[5] = {1, 2, 3, 4, 5};
queue->write(data, 5); // Write 5 items
```

- **Read:**

```cpp
uint16_t output[5];
readerQueue.read(output, 5);  // Read 5 items
```

- **Blocking Versions:**
  - Wait until data is available or space is free.

```cpp
queue->writeBlocking(data, 5, 1'000'000); // 1ms timeout
readerQueue.readBlocking(output, 5, 1'000'000);
```

🚦 EventFlag

Think of **EventFlag** as a "doorbell" between writer and reader:

- Writer rings when data is written.
- Reader waits for the bell before reading.
- Helpful for blocking reads/writes.

💡 Zero-Copy = Superfast Transfer

Normally FMQ uses `memcpy`, which takes time.
Zero-copy lets you write/read directly into the buffer like this:

```cpp
MessageQueue::MemTransaction tx;
if (queue->beginWrite(5, &tx)) {
    auto& first = tx.getFirstRegion();
    memcpy(first.getAddress(), yourData, first.getLengthInBytes());
    queue->commitWrite(5);
}
```

No copy = ultra-fast 🚀.

⚠️ Limits to Remember

- FMQ size is fixed.
- Too many items or too slow reader = data loss (in unsynchronized).
- Readers should keep up or reset to avoid overflow errors.

🧠 Real-Life Example

Camera HAL → App:

- Camera hardware continuously sends frames (images) to the app.
- App reads these frames via **FMQ** for real-time display or processing.
- FMQ keeps the pipeline smooth and low-latency.

Would you like a **C++ working code sample** for this (writer + reader)?

13. ↪
Unsynchronized1ManyYesYes**Synchronized**11 (Only one)NoYes
explain the overflow with example

Let's understand **FMQ overflow in unsynchronized queues** with a **very simple example**.

🧠 What is Overflow?

In **unsynchronized FMQ**, there is:

- **1 writer**
- **Many readers**
- Each reader has its own read pointer.
- If a reader is too **slow**, and the **writer keeps writing**, eventually the data the reader needs will be **overwritten** — that's called **overflow**.

📘 Real-world Analogy

- A **radio DJ (writer)** keeps announcing lottery numbers every second.
- Listeners (readers) tune in and note down numbers.
- But one listener is **slow** and tries to note down older numbers.
- The DJ already moved on, and the older numbers are **gone** (overwritten).
- Listener gets a message: "Too late, reset to current list."

💻 FMQ Example (in C++-style logic)

Let's say:

- FMQ has **capacity = 4**
- Writer keeps writing `W1, W2, W3, W4, W5, W6, ...`
- Reader is trying to read old values like `W1` or `W2`, but they were **already overwritten**.

✅ Normal Flow

```cpp
Writer writes: W1, W2, W3, W4
Reader reads:  W1, W2, W3, W4
OK 👍
```

❌ Overflow Case

```cpp
Writer writes: W1, W2, W3, W4  ← Queue is now full
Reader hasn't read anything yet

Writer writes: W5  ← W1 is removed (overwritten)
Writer writes: W6  ← W2 is removed
Writer writes: W7  ← W3 is removed
Writer writes: W8  ← W4 is removed

Reader now tries to read W1 → ❌ OVERFLOW
```

🚨 What Happens Now?

- Reader **can't get W1** anymore — it's lost.
- FMQ detects overflow.

- It **resets the reader** to the current write position **minus half the queue size**.
- That gives reader a chance to **catch up** again.

Example: Queue capacity = 4

```cpp
Write pointer = 8
→ Reader’s pointer is reset to 8 - 2 = 6
→ Reader starts reading from W7 next time
```

So the reader skips W5 and W6 too — to leave room for new incoming data.

🔧 Code-like Reaction

```cpp
if (!readerQueue.read(data, 1)) {
    // read failed due to overflow
    // FMQ resets the reader's read pointer to write pointer - (capacity / 2)
}
```

📝 Key Rules to Remember

Rule	Explanation
Only 1 writer	Keeps writing into the ring buffer
Many readers	Each reader reads at their own speed
Queue size is fixed	Example: 4 elements
If writer wraps and reader is slow	Older data gets overwritten
FMQ handles overflow	Reader is reset automatically
Overflow is invisible unless checked	**`availableToRead()`** > capacity means overflow happened

---

Would you like a **visual diagram** for this ring buffer overflow behavior?

14. is FMQ zero-copy?

Yes, **FMQ (Fast Message Queue)** in Android **can support zero-copy**, **but only if you use specific APIs for it**.

✅ What is Zero-Copy?

Zero-copy means data is **not duplicated** (via **`memcpy`**) between processes — instead, both the sender and receiver **access the same shared memory** directly.

In typical IPC (e.g., Binder), data is copied from:

1. Sender memory → Kernel buffer
2. Kernel buffer → Receiver memory
   ⬆️ This involves 2 copies.

But with FMQ **zero-copy** APIs:

- Both sides read/write **directly from shared memory**
- ✅ No memcpy
- ✅ Better performance (especially for high-frequency data like sensor or camera)

✅ How to use FMQ as Zero-Copy

FMQ provides special APIs:

```cpp
// Writer side
bool beginWrite(size_t count, MemTransaction* tx);
bool commitWrite(size_t count);

// Reader side
bool beginRead(size_t count, MemTransaction* tx);
bool commitRead(size_t count);
```

These give you **pointers into the FMQ ring buffer** directly, so you can:

- Write your data directly
- Or read it directly

✅ This avoids copying data between intermediate buffers.

📝 Simple Example

```cpp
// Writer side
MessageQueueSync::MemTransaction tx;
if (fmq->beginWrite(3, &tx)) {
    auto region = tx.getFirstRegion();
    uint8_t* buf = region.getAddress();
    buf[0] = 42;
    buf[1] = 43;
    buf[2] = 44;
    fmq->commitWrite(3);
}
```

No need to do **`memcpy`** — you write directly to shared memory.

❗ Important Notes

Point	Description
Zero-copy supported only in C++	No Java support for now

Point	Description
Needs Android 8.0+	Introduced with HIDL in O
Supported in HIDL-based FMQ only	AIDL FMQ has similar support but with different APIs
Cannot use nested types	No **`string`** or **`vec`** inside FMQ element type

---

🚫 Not Zero-Copy By Default

- Normal `write()` / `read()` APIs in FMQ **use memcpy** — **not zero-copy**
- You **must** use `beginWrite()/commitWrite()` or `beginRead()/commitRead()` to get zero-copy behavior

Would you like a **diagram** of how FMQ zero-copy works vs. normal IPC copy?

15. show the example with HDIL for this

Here's a **complete example** that shows how to implement **FMQ zero-copy using HIDL** in C++. This includes:

1. HIDL `.hal` interface definition with FMQ
2. Server-side (HAL) code that writes using **zero-copy**
3. Client-side code that reads using **zero-copy**

🧩 1. HIDL Interface Definition

`IMyHal.hal` (place under **`vendor/mycompany/hardware/myhal/1.0/`**):

```hidl
package vendor.mycompany.hardware.myhal@1.0;

import android.hidl.safe_union@1.0; // Optional
import android.hidl.memory@1.0;

interface IMyHal {
    // Pass FMQ to client
    getFmqDescriptor() generates (fmq_sync<uint8_t> myQueue);
}
```

🏗️ 2. Build Setup

Your **`Android.bp`** file for the HAL:

```bp
hidl_interface {
    name: "vendor.mycompany.hardware.myhal@1.0",
    root: "vendor",
    srcs: ["IMyHal.hal"],
    interfaces: [
        "android.hidl.base@1.0",
    ],
    gen_java: false,
}
```

⚙️ 3. Server (HAL) Implementation

Here's how to create and write to an FMQ using **zero-copy**:

```cpp
#include "MyHal.h"
#include <fmq/MessageQueue.h>

using ::android::hardware::MessageQueue;
using ::android::hardware::kSynchronizedReadWrite;
using vendor::mycompany::hardware::myhal::V1_0::IMyHal;
using android::hardware::Return;

constexpr size_t kQueueSize = 128;

struct MyHalImpl : public IMyHal {
    std::unique_ptr<MessageQueue<uint8_t, kSynchronizedReadWrite>> mFmq;

    MyHalImpl() {
        mFmq = std::make_unique<MessageQueue<uint8_t, kSynchronizedReadWrite>>(kQueueSize);
    }

    Return<void> getFmqDescriptor(getFmqDescriptor_cb _hidl_cb) override {
        if (!mFmq || !mFmq->isValid()) return Void();
        _hidl_cb(*mFmq->getDesc());
        return Void();
    }

    void writeDataToFmq() {
        const size_t count = 4;

        MessageQueue<uint8_t, kSynchronizedReadWrite>::MemTransaction tx;
```

```
        if (mFmq->beginWrite(count, &tx)) {
            auto first = tx.getFirstRegion();
            auto second = tx.getSecondRegion();

            std::memset(first.getAddress(), 0x42, first.getLength());
            std::memset(second.getAddress(), 0x43, second.getLength());

            mFmq->commitWrite(count);
        } else {
            ALOGE("FMQ write failed");
        }
    }
};
```

📱 4. Client Implementation

Client creates the **`MessageQueue`** from the descriptor:

```cpp
#include <vendor/mycompany/hardware/myhal/1.0/IMyHal.h>
#include <fmq/MessageQueue.h>

using ::vendor::mycompany::hardware::myhal::V1_0::IMyHal;
using ::android::hardware::MessageQueue;
using ::android::hardware::kSynchronizedReadWrite;

sp<IMyHal> myHal = IMyHal::getService();
if (myHal == nullptr) {
    ALOGE("Failed to get service");
    return;
}

IMyHal::getFmqDescriptor_cb cb = [&](const MessageQueue<uint8_t, kSynchronizedReadWrite>::Descriptor& desc) {
    auto fmq = std::make_unique<MessageQueue<uint8_t, kSynchronizedReadWrite>>(desc, /* resetPointers = */ true);

    if (!fmq->isValid()) {
        ALOGE("Invalid FMQ");
        return;
    }

    const size_t count = 4;
    MessageQueue<uint8_t, kSynchronizedReadWrite>::MemTransaction tx;
    if (fmq->beginRead(count, &tx)) {
        auto first = tx.getFirstRegion();
        auto second = tx.getSecondRegion();

        for (size_t i = 0; i < first.getLength(); ++i)
            printf("Byte %zu: %02x\n", i, first.getAddress()[i]);

        for (size_t i = 0; i < second.getLength(); ++i)
            printf("Byte %zu: %02x\n", i + first.getLength(), second.getAddress()[i]);

        fmq->commitRead(count);
    }
};

myHal->getFmqDescriptor(cb);
```

🧠 Summary

Part	Code
Interface	Uses **`fmq_sync<uint8_t>`**
Server	Writes using **`beginWrite()`** + **`commitWrite()`**
Client	Reads using **`beginRead()`** + **`commitRead()`**
Zero-copy	Achieved by directly accessing **`getAddress()`** of **`MemRegion`**

Would you like me to generate a **full Android Studio C++ project** or a **skeleton AOSP integration** for this HAL?

16. so here no is binder used?

Excellent question.

👉 **Yes, Binder is used** — but only once — to **send the FMQ descriptor** from the HAL (server) to the client.

After that, **Binder is no longer involved in actual data transfer.** Let me explain:

🔁 Binder vs FMQ in HIDL

Operation	Binder used?	Why
Calling **`getFmqDescriptor()`**	✅ Yes	Because this is a normal HIDL RPC method call
Sending or receiving actual data via FMQ	❌ No	Because FMQ uses **shared memory + atomic pointers** instead of IPC

📦 What exactly happens?

1. **Binder is used once** to send the **`MQDescriptorSync`** or **`MQDescriptorUnsync`** from server to client.

   This descriptor contains:

   - File descriptor (FD) to shared memory (ashmem)
   - Queue size and element type
   - Read/write positions and optional event flag word

2. **FMQ ring buffer** is set up in shared memory (ashmem or mmap).
   - Both processes can access it directly (no kernel context switch).
   - One writes to it, the other reads from it.
3. Future read/write operations happen using:
   - Direct memory access (zero-copy)
   - Atomics for read/write positions
   - Optional **`EventFlag`** to wait without polling

🧠 Why is this better?

Because **Binder is slower** for large or frequent data, since:

- It requires kernel-mode transitions
- It does **`memcpy`** across processes
- It can block and incur scheduling cost

FMQ avoids all of this.

✅ Final Answer:

> **Binder is used only once** to send the FMQ descriptor during **`getFmqDescriptor()`**. After that, all data is sent **outside of Binder** using shared memory and atomics — making FMQ an **efficient, low-latency, zero-copy communication mechanism.**

Would you like a diagram to show the memory layout and flow?

17. Use binder IPC

bookmark_border
This page describes changes to the binder driver in Android 8, provides details on using binder IPC, and lists required SELinux policy.

Changes to binder driver
Starting in Android 8, the Android framework and HALs now communicate with each other using binder. As this communication dramatically increases binder traffic, Android 8 includes several improvements designed to keep binder IPC fast. SoC vendors and OEMs should merge directly from the relevant branches of android-4.4, android-4.9, and higher of the kernel/common project.

Multiple binder domains (contexts)
Common-4.4 and higher, including upstream
To cleanly split the binder traffic between framework (device-independent) and vendor (device-specific) code, Android 8 introduced the concept of a binder context. Each binder context has its own device node and its own context (service) manager. You can access the context manager only through the device node to which it belongs and, when passing a binder node through a certain context, it is accessible from that same context only by another process, thus completely isolating the domains from each other. For details on using, see vndbinder and vndservicemanager.

Scatter-gather
Common-4.4 and higher, including upstream
In previous releases of Android, every piece of data in a binder call was copied three times:

Once to serialize it into a Parcel in the calling process
Once in the kernel driver to copy the Parcel to the target process
Once to unserialize the Parcel in the target process
Android 8 uses scatter-gather optimization to reduce the number of copies from 3 to 1. Instead of serializing data in a Parcel first, data remains in its original structure and memory layout and the driver immediately copies it to the target process. After the data is in the target process, the structure and memory layout is the same and the data can be read without requiring another copy.

Fine-grained locking
Common-4.4 and higher, including upstream
In previous Android releases, the binder driver used a global lock to protect against concurrent access to critical data structures. While there was minimal contention for the lock, the main problem was that if a low-priority thread obtained the lock and then got preempted, it could seriously delay higher-priority threads needing to obtain the same lock. This caused jank in the platform.

Initial attempts to resolve this problem involved disabling preemption while holding the global lock. However, this was more of a hack than a true solution, and was eventually rejected by upstream and discarded. Subsequent attempts focused on making locking more fine-grained, a version of which has been running on Pixel devices since January 2017. While the majority of those changes were made public, substantial improvements were made in subsequent versions.

After identifying small issues in the fine-grained locking implementation, we devised an improved solution with a different locking architecture and submitted the changes in all common kernel branches. We continue to test this implementation on a large number of different devices; as we are unaware of any outstanding issues, this is the recommended implementation for devices shipping with Android 8.

Note: We strongly encourage budgeting sufficient testing hours for fine-grained locking.
Real-time priority inheritance
Common-4.4 and common-4.9 (upstream coming soon)
The binder driver has always supported nice priority inheritance. As an increasing number of processes in Android run at real-time priority, in some cases it now makes sense that if a real-time thread makes a binder call, the thread in the process that handles that call also runs at real-time priority. To support these use cases, Android 8 now implements real-time priority inheritance in the binder driver.

In addition to transaction-level priority inheritance, node priority inheritance allows a node (binder service object) to specify a minimum priority at which calls into this node should be executed. Previous versions of Android already supported node priority inheritance with nice values, but Android 8 adds support for real-time scheduling policies node inheritance.

Note: The Android performance team found that real-time priority inheritance caused unwanted side-effects in the framework binder domain (/dev/binder), so real-time priority inheritance is disabled for that domain.
Userspace changes
Android 8 includes all userspace changes required to work with the current binder driver in the common kernel with one exception: The original implementation to disable real-time priority inheritance for /dev/binder used an ioctl. Subsequent development switched control of priority inheritance to a more fine-grained method that is per binder mode (and not per context). Thus, the ioctl isn't in the Android common branch and is instead submitted in our common kernels.

The effect of this change is that real-time priority inheritance is disabled by default for every node. The Android performance team has found it beneficial to enable real-time priority inheritance for all nodes in the hwbinder domain. To achieve that same effect, cherry-pick this change in userspace.

SHAs for common kernels
To obtain necessary changes to the binder driver, sync to the appropriate SHA:

Common-3.18
cc8b90c121de ANDROID: binder: don't check prio permissions on restore.
Common-4.4
76b376eac7a2 ANDROID: binder: don't check prio permissions on restore.
Common-4.9
ecd972d4f9b5 ANDROID: binder: don't check prio permissions on restore.
Work with binder IPC
Historically, vendor processes have used binder interprocess communication (IPC) to communicate. In Android 8, the /dev/binder device node becomes exclusive to framework processes, meaning vendor processes no longer have access to it. Vendor processes can access /dev/hwbinder, but must convert their AIDL interfaces to use HIDL. For vendors who want to continue using AIDL interfaces between vendor processes, Android supports binder IPC as described below. In Android 10, Stable AIDL allows all processes to use /dev/binder while also solving for the stability guarantees HIDL and /dev/hwbinder solved. For how to use Stable AIDL, see AIDL for HALs.

vndbinder
Warning: Devices launching with Android 11 or later which need to use these features must explicitly opt into using vndservicemanager by specifying PRODUCT_PACKAGES += vndservicemanager. This functionality is deprecated as of Android 11. The recommendation is to remove it, use the NDK and Rust backends of AIDL as this saves a process and extra threadpools in Android.
Android 8 supports a new binder domain for use by vendor services, accessed using /dev/vndbinder instead of /dev/binder. With the addition of /dev/vndbinder, Android now has the following three IPC domains:

IPC Domain	Description
/dev/binder	IPC between framework/app processes with AIDL interfaces
/dev/hwbinder	IPC between framework/vendor processes with HIDL interfaces
IPC between vendor processes with HIDL interfaces
/dev/vndbinder	IPC between vendor/vendor processes with AIDL Interfaces
For /dev/vndbinder to appear, ensure the kernel configuration item CONFIG_ANDROID_BINDER_DEVICES is set to "binder,hwbinder,vndbinder" (this is the default in Android's common kernel trees).

Normally, vendor processes don't open the binder driver directly and instead link against the libbinder userspace library, which opens the binder driver. Adding a method for ::android::ProcessState() selects the binder driver for libbinder. Vendor processes should call this method before calling into ProcessState, IPCThreadState, or before making any binder calls in general. To use, place the following call after the main() of a vendor process (client and server):

```
ProcessState::initWithDriver("/dev/vndbinder");
```

vndservicemanager
Previously, binder services were registered with servicemanager, where they could be retrieved by other processes. In Android 8, servicemanager is now used exclusively by framework and app processes and vendor processes can no longer access it.

However, vendor services can now use vndservicemanager, a new instance of servicemanager that uses /dev/vndbinder instead of /dev/binder and which is built from the same sources as framework servicemanager. Vendor processes don't need to make changes to talk to vndservicemanager; when a vendor process opens /dev/vndbinder, service lookups automatically go to vndservicemanager.

The vndservicemanager binary is included in Android's default device makefiles.

SELinux policy
Vendor processes that want to use binder functionality to communicate with each other need the following:

Access to /dev/vndbinder.
Binder {transfer, call} hooks into vndservicemanager.
binder_call(A, B) for any vendor domain A that wants to call into vendor domain B over the vendor binder interface.
Permission to {add, find} services in vndservicemanager.
To fulfill requirements 1 and 2, use the vndbinder_use() macro:

```
vndbinder_use(some_vendor_process_domain);
```

To fulfill requirement 3, the binder_call(A, B) for vendor processes A and B that need to talk over binder can stay in place, and doesn't need renaming.

To fulfill requirement 4, you must make changes in the way service names, service labels, and rules are handled.

For details on SELinux, see Security-Enhanced Linux in Android. For details on SELinux in Android 8.0, see SELinux for Android 8.0.

Service names
Previously, vendor processes registered service names in a service_contexts file and added corresponding rules for accessing that file. Example service_contexts file from device/google/marlin/sepolicy:

```
AtCmdFwd                      u:object_r:atfwd_service:s0
cneservice                    u:object_r:cne_service:s0
qti.ims.connectionmanagerservice     u:object_r:imscm_service:s0
rcs                           u:object_r:radio_service:s0
uce                           u:object_r:uce_service:s0
vendor.qcom.PeripheralManager        u:object_r:per_mgr_service:s0
```

In Android 8, vndservicemanager loads the vndservice_contexts file instead. Vendor services migrating to vndservicemanager (and which are already in the old service_contexts file) should be added to the new vndservice_contexts file.

Service labels
Previously, service labels such as u:object_r:atfwd_service:s0 were defined in a service.te file. Example:

```
type atfwd_service,      service_manager_type;
```

In Android 8, you must change the type to vndservice_manager_type and move the rule to the vndservice.te file. Example:

```
type atfwd_service,      vndservice_manager_type;
```

servicemanager rules
Previously, rules granted domains access to add or find services from servicemanager. Example:

```
allow atfwd atfwd_service:service_manager find;
allow some_vendor_app atfwd_service:service_manager add;
```

In Android 8, such rules can stay in place and use the same class. Example:

```
allow atfwd atfwd_service:service_manager find;
allow some_vendor_app atfwd_service:service_manager add;
```

Let's break this down **in simple words** — this is about how **Binder IPC changed in Android 8 and later**, especially for **vendor vs framework** components.

🧠 What is Binder IPC?

Binder is Android's main way of letting different processes (apps, services, HALs) talk to each other — it's an **IPC (Inter-Process Communication)** mechanism.

🔁 What changed in Android 8?

Android 8 split the Binder system into **multiple domains** to **separate vendor code from system/framework code**.

📁 Domains (Binder contexts):

Binder device	Used for	Language
`/dev/binder`	Apps and framework services	AIDL
`/dev/hwbinder`	Framework <-> HAL	HIDL
`/dev/vndbinder`	Vendor <-> Vendor	AIDL (internal to vendor code)

> 🔐 Purpose: Prevent accidental mixing of vendor and framework components, which helps with **modularity and stability**.

🔧 Why separate domains?

- Before Android 8: Everyone used **`/dev/binder`**, including HALs.
- After Android 8:
  - Framework uses **`/dev/binder`**
  - HALs and vendors use **`/dev/hwbinder`** and **`/dev/vndbinder`**
- Each has its **own servicemanager**:
  - **`servicemanager`** → for framework
  - **`vndservicemanager`** → for vendor

📦 How to use `/dev/vndbinder` in vendor processes?

Step 1: Choose correct binder device

At the top of your **`main()`** function in C++ vendor process:

```cpp
android::ProcessState::initWithDriver("/dev/vndbinder");
```

> This tells the process to use the vendor binder driver.

Step 2: Register and access services

Services are registered with **`vndservicemanager`** (automatically handled when using **`/dev/vndbinder`**).

You can use:

- **`addService()`** to register
- **`getService()`** to fetch

Same APIs as before, just running in vendor space now.

🔐 SELinux rules needed

If your vendor services use binder, you need to:

1. Allow access to **`/dev/vndbinder`**:

```bash
vndbinder_use(my_vendor_service_domain)
```

2. Allow calling from one vendor domain to another:

```bash
binder_call(vendor_service_A, vendor_service_B)
```

3. Move service labels:

- From **`service_contexts`** → to **`vndservice_contexts`**
- Label types change from **`service_manager_type`** → **`vndservice_manager_type`**

4. Service add/find rules stay the same:

```bash
allow myservice mylabel:service_manager { add find };
```

🏎️ Performance Upgrades in Binder (Android 8)

These are kernel-level improvements:

- ✅ **Scatter-gather**: Only 1 copy needed instead of 3 when sending data over Binder.
- ✅ **Fine-grained locking**: Avoids delays when multiple threads use Binder.
- ✅ **Real-time priority inheritance**: Lets real-time threads stay responsive.

🧪 Real example

Suppose you are a vendor writing a HAL service like this:

- Server: `vendor.mycompany.foo@1.0::IFooService`
- Client: Another vendor component, not the framework

You can:

- Register the service using `registerAsService()` (HIDL → uses `/dev/hwbinder`)
- Or use AIDL with `/dev/vndbinder` for vendor-only processes using `ProcessState::initWithDriver("/dev/vndbinder")`

  Android recommends **using AIDL +** `/dev/vndbinder` for vendor-only services after Android 10+.

✅ Summary

Feature	Purpose
`/dev/binder`	Apps and framework
`/dev/hwbinder`	HIDL between framework and HAL
`/dev/vndbinder`	AIDL between vendor processes
`vndservicemanager`	Service registry for `/dev/vndbinder`
`initWithDriver()`	Chooses which binder device to use
`vndbinder_use()`	SELinux rule to allow use of `/dev/vndbinder`
HIDL → `/dev/hwbinder`	Binder used once to send FMQ descriptor, data transferred without Binder

---

If you'd like, I can show:

- Code examples for registering/using services in `/dev/vndbinder`
- How to define a HIDL/AIDL service to use binder/FMQueue properly

Let me know!

18. ↪
Each has its **own servicemanager**: * `servicemanager` → for framework * `vndservicemanager` → for vendor
explain this

Sure! Let's explain this part in **simple words**:

🧠 What is a `servicemanager`?

In Android, when a service is created (like a HAL or system service), it **registers itself** so that **other processes can find and talk to it**.

The thing that **keeps track of all these services** is called the `servicemanager`.

✅ What changed?

Before Android 8:

- There was **only one** `servicemanager`.
- Both **system services** (like CameraService) and **vendor HALs** (like your custom LED HAL) **used the same one**.

After Android 8:

> To **separate vendor and system**, Android introduced **two** `servicemanagers`, each for a different world:

Binder Device	Who Uses It	Servicemanager
`/dev/binder`	Apps & Android Framework	`servicemanager`
`/dev/vndbinder`	Vendor <-> Vendor	`vndservicemanager`

📦 So what happens now?

- If you're in a **system or app process**, you use `/dev/binder`, and when you do something like `getService("camera")`, it talks to the **system** `servicemanager`.
- If you're in a **vendor service or HAL**, and you use `/dev/vndbinder`, then all service registration (`addService`) and lookups (`getService`) **go through** `vndservicemanager`.

🔍 Analogy

Think of it like this:

- `servicemanager`: A **reception desk for Android Framework** services
- `vndservicemanager`: A **separate reception desk for Vendor services**

They keep **separate books**. Vendor can't register or look up services from the framework's desk, and vice versa.

💡 Why?

Because of **Project Treble** — Android needed to **separate the vendor layer from the framework layer** so that vendors could update HALs without needing to rebuild the Android framework.

✅ Summary

Service	Handles registration/lookup for
`servicemanager`	Android Framework & Apps
`vndservicemanager`	Vendor services

- Each Binder domain has its own `servicemanager`.
- Services using `/dev/vndbinder` automatically talk to `vndservicemanager`.

Let me know if you want to see **how to register a vendor service in vndservicemanager** or **SELinux changes needed**.

19. and hwservicemanager

Great question!

Let's now break down what **`hwservicemanager`** is — and how it fits in with **`servicemanager`** and **`vndservicemanager`**.

🔧 What is `hwservicemanager`?

- It is the **service registry for HIDL-based HALs**.
- HIDL = HAL Interface Definition Language (used before AIDL for HALs).
- When a **HIDL HAL** (like **`vendor.mycompany.led@1.0::ILed`**) is created, it **registers itself with** **`hwservicemanager`**.
- When an app or system component wants to use that HAL, it asks **`hwservicemanager`** to get the HAL.

💡 Why do we need `hwservicemanager`?

Because HIDL-based HALs **don't use regular Binder IPC** (**`/dev/binder`**).
They use a **separate Binder domain**: **`/dev/hwbinder`**.

To support this, Android introduced a **new registry process**:
🔧 **`hwservicemanager`** — the "servicemanager" for HALs that use HIDL.

🔄 Recap: Who uses what?

Purpose	Used For	Uses Binder Node	Registry (Manager)
Android Framework/App	AIDL, framework services	`/dev/binder`	`servicemanager`
Vendor–Vendor AIDL	AIDL between vendor procs	`/dev/vndbinder`	`vndservicemanager`
Framework–Vendor HAL	HIDL HAL (like camera HAL)	`/dev/hwbinder`	`hwservicemanager`

🧪 Example

Let's say we have a HIDL HAL:

```hal
package vendor.mycompany.led@1.0;
interface ILed {
  setState(bool on);
};
```

1. The **vendor HAL implementation** runs and calls:

   ```cpp
   ledService->registerAsService();  // registers with hwservicemanager
   ```

2. The **Android framework or vendor app** calls:

   ```cpp
   sp<ILed> service = ILed::getService(); // internally uses hwservicemanager
   ```

3. **`hwservicemanager`** looks up the service and gives you the connection.

🔐 Security domains

Binder Device	Used by...	Manager
`/dev/binder`	Framework + apps	`servicemanager`
`/dev/vndbinder`	Vendor-only AIDL	`vndservicemanager`
`/dev/hwbinder`	HIDL-based HALs	`hwservicemanager`

✅ Summary

- `hwservicemanager` = Service manager for **HIDL HALs**
- Used over `/dev/hwbinder`
- It's **always used when HIDL HALs are involved** (before Android moved to AIDL HALs)
- Works alongside `servicemanager` and `vndservicemanager`, but serves a **different purpose**

Let me know if you want to see:

- Code example using `hwservicemanager`
- Internals (how `registerAsService()` works)
- How to implement a HIDL HAL end to end

20. ↪
🔐 SELinux rules needed ---------------------- If your vendor services use binder, you need to: 1. Allow access to `/dev/vndbinder`: bash CopyEdit `vndbinder_use(my_vendor_service_domain)` 2. Allow calling from one vendor domain to another: bash CopyEdit `binder_call(vendor_service_A, vendor_service_B)` 3. Move service labels: * From `service_contexts` → to `vndservice_contexts` * Label types change from `service_manager_type` → `vndservice_manager_type` 4. Service add/find rules stay the same: bash CopyEdit `allow myservice mylabel:service_manager { add find };` explain in simple words

Sure! Let me explain that part in **simple words** with a real-world analogy.

🔐 What is this about?

This is about setting **SELinux permissions** so that **your vendor services** (custom services made by hardware vendors like Samsung, Qualcomm, etc.) can **talk to each other** using **Binder IPC**.

In Android 8 and above, **vendor services use `/dev/vndbinder`**, not the regular **`/dev/binder`**.

🎯 So what do you need to do?

You need to **tell SELinux**:

1. 📂 "My service can use `/dev/vndbinder`"

```bash
vndbinder_use(my_vendor_service_domain)
```

🔍 This line means:

> "Hey SELinux, my vendor service (e.g., **`camera_daemon`**) is allowed to use the vendor binder device (**`/dev/vndbinder`**)."

---

2. 🔁 "This service can call that service"

```bash
binder_call(service_A, service_B)
```

🔍 This line means:

> "Allow service_A to send Binder messages to service_B."

📦 Example:
If your **GPS HAL** wants to call your **Sensor HAL**, you add:

```bash
binder_call(gps_service, sensor_service)
```

---

3. 🏷️ Move and rename your service labels

In older Android versions, service names were listed in **`service_contexts`**.
But in Android 8+, vendor services must move to **`vndservice_contexts`**.

You also rename types like this:

Old	New
`service_manager_type`	**`vndservice_manager_type`**

---

4. ✅ Add rules for finding or registering a service

```bash
allow myservice mylabel:service_manager { add find };
```

🔍 This line means:

> "Allow this service to **register itself** (add) or **find another service** (find) through the **`vndservicemanager`**."

📦 Summary

In simple terms:

> "You're telling Android security:
>
> - My service is safe to use vendor Binder (**`/dev/vndbinder`**)
> - It can talk to other vendor services
> - Here's its label
> - Let it register and find other services"

Want me to show how the **`vndservice_contexts`** file looks or what a full **`.te`** rule file would be like?

21. ↪
🏎️ Performance Upgrades in Binder (Android 8) -- These are kernel-level improvements: * ✅ **Scatter-gather**: Only 1 copy needed instead of 3 when sending data over Binder. * ✅ **Fine-grained locking**: Avoids delays when multiple threads use Binder. * ✅ **Real-time priority inheritance**: Lets real-time threads stay responsive.
explain in simple words

Sure! Let's break these **Binder performance upgrades** in **Android 8** into **simple words** with examples:

🚀 Why were upgrades needed?

Binder is Android's way for processes (apps, system services, HALs) to **talk to each other**. But in Android 8, this traffic increased a lot, so Google improved how Binder works inside the Linux kernel.

🛠️ Performance Upgrades

1. ✅ Scatter-Gather = Less Copying

Before (Android < 8):

- When you send data using Binder, it's copied **3 times**:
  - App → Parcel (serialize)
  - Parcel → kernel
  - Kernel → receiver

After (Android 8+):

- Data is only copied **once** using **scatter-gather**.
- It's like handing someone a book **directly** instead of photocopying it 3 times.

🎯 **Result**: Speeds up data transfer.

---

2. ✅ Fine-Grained Locking = Less Waiting

Before:

- Binder used **one big lock** for everything.
- If a **slow thread** grabbed the lock, faster ones had to wait.

After:

- Uses **many small locks** instead of one.
- Threads don't block each other unnecessarily.

🎯 **Result**: Less lag, smoother performance.

---

3. ✅ Real-Time Priority Inheritance = Smooth for Time-Critical Services

Before:

- A real-time thread (e.g. audio, camera) could get **stuck waiting** on a lower-priority thread.

After:

- If a high-priority thread calls a service, that service **temporarily runs with higher priority** too.

🎯 **Result**: Real-time services (like voice call, camera) stay responsive.

🧠 Think of it like this:

If Binder was a **delivery truck**, then:

Feature	Analogy
Scatter-Gather	Handing the box directly (no repacking)
Fine-Grained Locking	Multiple delivery gates instead of one
Real-Time Inheritance	Giving ambulance priority to critical trucks

---

Let me know if you want code or visuals to go along with this!

22. so now binder is zero-copy?

Great question!

🔍 So, is Binder zero-copy in Android 8 and above?

Not completely zero-copy, but it's **closer to it.** Let's explain:

📦 Before Android 8:

Binder did **3 copies** for every transaction:

1. Serialize in the sending process.
2. Kernel copies data to receiving process.
3. Deserialize in receiving process.

🚀 Android 8 introduced Scatter-Gather I/O:

- It **skips the first and third copies.**
- The data stays in its **original memory layout**.
- Binder driver maps this memory **directly** into the receiving process.

➡️ So only **one copy happens** — from sender's memory to receiver's.

🟢 So is it zero-copy?

No, not true zero-copy — because:

- One memory copy still happens (by the kernel).
- But **it avoids serialization and deserialization**, so it's **much faster**.

🔁 True Zero-Copy Alternative?

If you want **true zero-copy**, use:

- **FMQ (Fast Message Queue)** with shared memory.
- **Ashmem** (older) or **AIDL SharedMemory** (newer).

These allow processes to **read/write directly** into shared memory — no copying at all.

✅ Summary

Method	Zero-Copy?	Notes
Binder	❌ One-copy (after Android 8)	Scatter-gather reduces overhead
FMQ	✅ Yes	Shared memory + sync if needed
Ashmem	✅ Yes	Shared memory, older method
AIDL + SharedMemory	✅ Yes	Newer AIDL-based shared memory

Let me know if you want a visual diagram or code for FMQ vs Binder!

23. HIDL MemoryBlock

bookmark_border
The HIDL MemoryBlock is an abstract layer built on hidl_memory, HIDL @1.0::IAllocator, and HIDL @1.0::IMapper. It is designed for HIDL services that have multiple memory blocks to share a single memory heap.

Performance improvements
Using MemoryBlock in apps can significantly reduce the number of mmap/munmap and user space segmentation faults, thus improving performance. For example:

Using per hidl_memory for each buffer allocation averages 238 us/1 allocation.
Using MemoryBlock and sharing a single hidl_memory averages 2.82 us/1 allocation.
Architecture
The HIDL MemoryBlock architecture includes HIDL services with multiple memory blocks sharing a single memory heap:

HIDL MemoryBlock

Figure 1. HIDL MemoryBlock architecture

Normal usage
This section provides an example of using MemoryBlock by first declaring the HAL then implementing the HAL.

Declare the HAL
For the following example IFoo HAL:

```
import android.hidl.memory.block@1.0::MemoryBlock;

interface IFoo {
    getSome() generates(MemoryBlock block);
    giveBack(MemoryBlock block);
};
```

The Android.bp is as follows:

```
hidl_interface {
    ...
    srcs: [
        "IFoo.hal",
    ],
    interfaces: [
        "android.hidl.memory.block@1.0",
        ...
};
```

Implement the HAL
To implement the example HAL:

Get the hidl_memory (for details, refer to HIDL C++).

```
#include <android/hidl/allocator/1.0/IAllocator.h>

using ::android::hidl::allocator::V1_0::IAllocator;
using ::android::hardware::hidl_memory;
...
  sp<IAllocator> allocator = IAllocator::getService("ashmem");
  allocator->allocate(2048, [&](bool success, const hidl_memory& mem)
  {
        if (!success) { /* error */ }
        // you can now use the hidl_memory object 'mem' or pass it
  }));
```

Make a HidlMemoryDealer instance with the acquired hidl_memory:

```
#include <hidlmemory/HidlMemoryDealer.h>

using ::android::hardware::HidlMemoryDealer
/* The mem argument is acquired in the Step1, returned by the ashmemAllocator->allocate */
sp<HidlMemoryDealer> memory_dealer = HidlMemoryDealer::getInstance(mem);
```

Allocate MemoryBlock, which is a struct defined with HIDL.

Example MemoryBlock:

```
struct MemoryBlock {
IMemoryToken token;
uint64_t size;
uint64_t offset;
};
```

Example using the MemoryDealer to allocate a MemoryBlock:

```
#include <android/hidl/memory/block/1.0/types.h>

using ::android::hidl::memory::block::V1_0::MemoryBlock;

Return<void> Foo::getSome(getSome_cb _hidl_cb) {
   MemoryBlock block = memory_dealer->allocate(1024);
   if(HidlMemoryDealer::isOk(block)){
       _hidl_cb(block);
   ...
```

Deallocate MemoryBlock:

```
Return<void> Foo::giveBack(const MemoryBlock& block) {
   memory_dealer->deallocate(block.offset);
...
```

Manipulate the data:

```
#include <hidlmemory/mapping.h>
#include <android/hidl/memory/1.0/IMemory.h>

using ::android::hidl::memory::V1_0::IMemory;

sp<IMemory> memory = mapMemory(block);
uint8_t* data =

static_cast<uint8_t*>(static_cast<void*>(memory->getPointer()));
```

Config Android.bp:

```
shared_libs: [
      "android.hidl.memory@1.0",

      "android.hidl.memory.block@1.0"

      "android.hidl.memory.token@1.0",
      "libhidlbase",
      "libhidlmemory",
```

Review the flow to determine if you need to lockMemory.

Normally, MemoryBlock uses reference count to maintain the shared hidl_memory which is mmap()-ed the first time one of its MemoryBlock instances is mapped and ismunmap()-ed when nothing refers to it. To keephidl_memoryalways mapped, you can uselockMemory, a RAII style object that keeps the correspondinghidl_memory `mapped throughout the lock lifecycle. Example:`

```
#include <hidlmemory/mapping.h>
```

```
sp<RefBase> lockMemory(const sp<IMemoryToken> key);
```

Extended usage

This section provides details about the extended usage of MemoryBlock.

Use reference count to manage MemoryBlock

In most situations, the most efficient way to use MemoryBlock is to explicitly allocate/deallocate. However, in complicated apps using reference count for garbage collection might be a better idea. To have reference count on MemoryBlock, you can bind MemoryBlock with a binder object, which helps to count the references and deallocate the MemoryBlock when the count decreases to zero.

Declare the HAL

When declaring the HAL, describe a HIDL struct that contains a MemoryBlock instance and an IBase:

```
import android.hidl.memory.block@1.0::MemoryBlock;

struct MemoryBlockAllocation {
    MemoryBlock block;
    IBase refcnt;
};
```

Use MemoryBlockAllocation to replace MemoryBlock and remove the method to give back MemoryBlock. It's deallocated by reference counting with MemoryBlockAllocation. Example:

```
interface IFoo {
    allocateSome() generates(MemoryBlockAllocation allocation);
};
```

Implement the HAL

Example of the service side implementation of the HAL:

```
class MemoryBlockRefCnt: public virtual IBase {
   MemoryBlockRefCnt(uint64_t offset, sp<MemoryDealer> dealer)
     : mOffset(offset), mDealer(dealer) {}
   ~MemoryBlockRefCnt() {
       mDealer->deallocate(mOffset);
   }
 private:
   uint64_t mOffset;
   sp<MemoryDealer> mDealer;
};

Return<void> Foo::allocateSome(allocateSome_cb _hidl_cb) {
    MemoryBlockAllocation allocation;
    allocation.block = memory_dealer->allocate(1024);
    if(HidlMemoryDealer::isOk(block)){
        allocation.refcnt= new MemoryBlockRefCnt(...);
        _hidl_cb(allocation);
```

Example of the client side implementation of the HAL:

```
ifoo->allocateSome([&](const MemoryBlockAllocation& allocation){
    ...
);
```

Attach and retrieve metadata

Some apps need additional data to bind with the allocated MemoryBlock. You can append and retrieve metadata using two methods:

If the app accesses the metadata as often as the block itself, append the metadata and pass them all in a struct. Example:

```
import android.hidl.memory.block@1.0::MemoryBlock;

struct MemoryBlockWithMetaData{
    MemoryBlock block;
    MetaDataStruct metaData;
};
```

If the app accesses the metadata much less frequently than the block, it is more efficient to pass the metadata passively with an interface.

Example:

import android.hidl.memory.block@1.0::MemoryBlock;

struct MemoryBlockWithMetaData{
    MemoryBlock block;
    IMetaData metaData;
};
Next, bind the metadata with the MemoryBlock using MemoryDealer. Example:

MemoryBlockWithMetaData memory_block;
memory_block.block = dealer->allocate(size);
if(HidlMemoryDealer::isOk(block)){
    memory_block.metaData = new MetaData(...);

Let's break down **HIDL MemoryBlock** in **simple words**, with examples and diagrams where needed.

🧠 What is HIDL MemoryBlock?

MemoryBlock is a tool that helps multiple buffers share **one large memory heap** instead of allocating memory for each buffer separately.

Think of it like this:

 Instead of giving each guest a **cup of water** (separate allocation),
we let them pour from a **common water tank** (shared heap) — much faster and easier to manage.

⚡ Why use MemoryBlock?

Using `hidl_memory` for each allocation is **slow** (~238 microseconds).
Using `MemoryBlock` is **faster** (~2.8 microseconds) because it avoids repeated memory mapping (mmap/munmap) and reduces crashes.

🧱 Key Components

Component	Purpose
`hidl_memory`	Represents shared memory (like a water tank).
`IAllocator`	Allocates shared memory (like getting a water tank).
`HidlMemoryDealer`	Manages small memory blocks from the shared memory.
`MemoryBlock`	A struct that describes a small block inside that memory.
`IMemoryToken`	Token used to track access to the block.

✅ Example Flow

Step 1: Allocate a big chunk of memory (hidl_memory)

```cpp
sp<IAllocator> allocator = IAllocator::getService("ashmem");
allocator->allocate(2048, [&](bool success, const hidl_memory& mem) {
    // Success → you have a shared memory region
});
```

Step 2: Wrap it in a memory dealer

```cpp
sp<HidlMemoryDealer> memory_dealer = HidlMemoryDealer::getInstance(mem);
```

Step 3: Allocate a small piece (MemoryBlock)

```cpp
MemoryBlock block = memory_dealer->allocate(1024);
```

Now this `block` has:

- Offset → where the block starts inside shared memory.
- Size → how big the block is.
- Token → to map and access memory safely.

Step 4: Access the memory

```cpp
sp<IMemory> memory = mapMemory(block);
uint8_t* data = static_cast<uint8_t*>(memory->getPointer());
```

Now you can read/write memory like a normal array:

```cpp
data[0] = 42;
```

🗑️ Step 5: Free the block when done

```cpp
memory_dealer->deallocate(block.offset);
```

Or better yet, use **reference counting** (see below).

🔁 Advanced: Use Reference Counting (Auto Cleanup)

Instead of manually calling `giveBack()`, you can return a struct like this:

```hal
struct MemoryBlockAllocation {
    MemoryBlock block;
    IBase refcnt; // When this object dies, it auto-cleans memory
};
```

On the server side:

```cpp
class MemoryBlockRefCnt: public IBase {
   ~MemoryBlockRefCnt() {
       mDealer->deallocate(mOffset);
   }
};
```

Now memory gets automatically freed when the client is done.

🎁 Add Metadata (Optional)

If you need to send extra info with memory:

Option 1: Bundle metadata directly

```hal
struct MemoryBlockWithMetaData {
    MemoryBlock block;
```

```
    MetaDataStruct metaData;
};
```

Option 2: Use a separate interface for metadata

```
hal

struct MemoryBlockWithMetaData {
    MemoryBlock block;
    IMetaData metaData;
};
```

Use this when the metadata is **rarely accessed**, so it doesn't bloat the memory block itself.

🧠 Summary (Analogy)

Item	Analogy
`hidl_memory`	Large water tank
`MemoryBlock`	One cup taken from the tank
`IAllocator`	Tool that gives you the tank
`HidlMemoryDealer`	Faucet managing how cups are poured
`IMemoryToken`	Label on your cup to identify it
`lockMemory()`	Keep the tank open during usage
`refcnt`	Garbage collector for the cup

Would you like a code template that shows both client and server usage of `MemoryBlock`?

24. Network stack configuration tools

bookmark_border
The Android operating system contains standard Linux networking utilities such as ifconfig, ip, and ip6tables. These utilities reside on the system image and enable configuration of the entire Linux networking stack. On devices running Android 7.x and lower, vendor code is allowed to call these binaries directly, which presents the following problems:

Because network utilities are updated in the system image, they don't provide a stable implementation.
The scope of the networking utilities is so broad it is difficult to evolve the system image while guaranteeing predictable behaviour.
On devices running Android 8.0 and higher, the vendor partition remains the same while the system partition receives an update. To achieve this, Android 8.0 provides the ability to define a stable, versioned interface while also using SELinux restrictions to keep the interdependency of vendor and system image to a known good set.

Vendors can use the platform-provided network configuration utilities to configure the Linux networking stack, but these utilities don't yet include a HIDL interface wrapper. To define such an interface, Android 8.0 includes the netutils-wrapper-1.0 tool.

Netutils wrapper
The netutils wrapper utility provides a subset of the Linux network stack configuration that isn't affected by system partition updates. Android 8.0 contains version 1.0 of the wrappers, which allows you to pass the same arguments as the wrapped utilities, installed in the system partition at /system/bin as follows:

```
u:object_r:system_file:s0         /system/bin/ip-wrapper-1.0 -> netutils-wrapper-1.0
u:object_r:system_file:s0         /system/bin/ip6tables-wrapper-1.0 -> netutils-wrapper-1.0
u:object_r:system_file:s0         /system/bin/iptables-wrapper-1.0 -> netutils-wrapper-1.0
u:object_r:system_file:s0         /system/bin/ndc-wrapper-1.0 -> netutils-wrapper-1.0
u:object_r:netutils_wrapper_exec:s0 /system/bin/netutils-wrapper-1.0
u:object_r:system_file:s0         /system/bin/tc-wrapper-1.0 -> netutils-wrapper-1.0
```

Symlinks show the networking utilities wrapped by the netutils wrapper, which include:

ip
iptables
ip6tables
ndc
tc
To use these utilities in Android 8.0 and higher, vendor implementations must adhere to the following rules:

Vendor processes must not execute /system/bin/netutils-wrapper-1.0 directly; attempts to do so results in error.
All utilities wrapped by netutils-wrapper-1.0 must be launched using their symlinks. For example, change the vendor code that did this before (/system/bin/ip <FOO> <BAR>) to /system/bin/ip-wrapper-1.0 <FOO> <BAR>.
Executing the wrappers without domain transition is prohibited in platform SELinux policy. This rule must not be changed and is tested against in the Android Compatibility Test Suite (CTS).
Executing the utilities directly (e.g., /system/bin/ip <FOO> <BAR>) from the vendor processes is also prohibited in the platform SELinux policies. This rule must not be changed and is tested against in CTS.
Any vendor domain (process) that needs to launch a wrapper must add the following domain transition rule in the SELinux policy: domain_auto_trans(VENDOR-DOMAIN-NAME, netutils_wrapper_exec, netutils_wrapper).
Note: For details on SELinux in Android 8.0 and higher, see Customizing SEPolicy in Android 8.0+.
Netutils wrapper filters
Wrapped utilities can be used to configure almost any aspect of the Linux networking stack. However, to ensure it is possible to maintain a stable interface and allow updates to the system partition, only certain combinations of command line arguments are allowed; other commands are rejected.

Vendor interfaces and chains
The wrapper has a concept of vendor interfaces. These are interfaces typically managed by vendor code, such as cellular data interfaces. Typically, other types of interfaces (such as Wi-Fi) are managed by the HALs and the framework. The wrapper recognizes vendor interfaces by name (using a regular expression) and allows vendor code to perform many operations on them. Currently, vendor interfaces are:

Interfaces whose names end in "oem" followed by a number, such as oem0 or r_oem1234.
Interfaces used by current SOC and OEM implementations, such as rmnet_data[0-9].
Names of interfaces that are typically managed by the framework (such as wlan0) are never vendor interfaces.

The wrapper has a similar concept of vendor chains. These are used in iptables commands and are also recognized by name. Currently, vendor chains:

Start with oem_.
Are used by current SOC and OEM implementations, e.g., chains starting in nm_ or qcom_.
Allowed commands
Currently allowed commands are listed below. Restrictions are implemented via a set of regular expressions on the executed command lines. For details, refer to system/netd/netutils_wrappers/NetUtilsWrapper-1.0.cpp.

ip
The ip command is used to configure IP addresses, routing, IPsec encryption, and a number of other network parameters. The wrapper allows the following commands:

Add and remove IP addresses from vendor-managed interfaces.
Configure IPsec encryption.
iptables and ip6tables
The iptables and ip6tables commands are used to configure firewalling, packet mangling, NAT, and other per-packet processing. The wrapper allows the following commands:

Add and delete vendor chains.
Add and delete rules in any chain that refers to packets going into (-i) or out of (-o) a vendor interface.
Jump to a vendor chain from any point in any other chain.
ndc
ndc is used to communicate to the netd daemon that performs most network configuration on Android. The wrapper allows the following commands:

Create and destroy OEM networks (oemXX).
Add vendor-managed interfaces to OEM networks.
Add routes to OEM networks.
Enable or disable IP forwarding globally and on vendor interfaces.
tc
The tc command is used to configure traffic queueing and shaping on vendor interfaces.

Here's a **simple explanation** of the Android **network stack configuration tools** and how they changed in Android 8.0+ using the **netutils-wrapper**:

🧠 Background (Android ≤ 7.x)

Before Android 8:

- Vendor code (in `/vendor`) directly called tools like `ifconfig`, `ip`, `iptables`, etc.
- These tools live in the `/system` partition.
- Problem: If the `/system` image is updated (e.g. in an OTA), vendor code might break because it relies on a specific version of those tools.

🔐 Problem in Android 8+

Starting Android 8:

- Google separated `/system` **(framework)** and `/vendor` **(device-specific code)** to allow independent updates.
- But vendor code still needed to configure networking. So...

✅ Solution: `netutils-wrapper-1.0`

Think of this as:

> 🔄 A **controlled gateway** between **vendor code** and **system-level networking tools** like `ip`, `iptables`, `ndc`, etc.

🛠️ How it Works

Instead of calling:

```sql
/system/bin/ip addr add ...
```

Vendors must call:

```sql
/system/bin/ip-wrapper-1.0 addr add ...
```

This wrapper does 2 things:

1. **Validates the command** (ensures it's safe and allowed).
2. **Executes it securely** inside system context.

📂 Wrapper Structure

Wrapper Binary	Points to →	What it wraps
`/system/bin/ip-wrapper-1.0`	→ `netutils-wrapper-1.0`	ip commands
`/system/bin/iptables-wrapper-1.0`	→ `netutils-wrapper-1.0`	iptables
`/system/bin/ndc-wrapper-1.0`	→ `netutils-wrapper-1.0`	ndc
`/system/bin/tc-wrapper-1.0`	→ `netutils-wrapper-1.0`	tc

They're just **symlinks**, all pointing to the **same binary**, but internally they handle different commands.

🔐 SELinux Requirements

To launch these wrappers from vendor code:

- You **can't call** `/system/bin/netutils-wrapper-1.0` directly.
- You **must use** the symlinks like `ip-wrapper-1.0`, `iptables-wrapper-1.0`, etc.
- SELinux must allow **domain transition** to `netutils_wrapper` domain:

  ```bash
  domain_auto_trans(VENDOR_DOMAIN, netutils_wrapper_exec, netutils_wrapper)
  ```

🚧 Allowed Commands

Only certain operations are allowed — others are **blocked**.

✔️ `ip`

- Add/remove IP addresses to vendor interfaces like `rmnet_data0`, `oem0`, etc.
- Configure IPsec.

✔️ `iptables` / `ip6tables`

- Add/delete **vendor chains** (e.g. `oem_nat`, `nm_firewall`, etc).
- Add/delete rules using `-i` or `-o` **vendor interfaces**.
- Jump to vendor chains from other chains.

✔️ `ndc`

- Create/destroy OEM networks (e.g. `oem0`, `oem1`)
- Add interfaces to OEM networks
- Add routes
- Enable IP forwarding

✔️ `tc`

- Configure traffic control (queueing, shaping) on vendor interfaces

---

🔍 What is a "vendor interface"?

Only specific interface names are treated as **vendor-owned**:

Examples:

- `rmnet_data0`
- `oem0`, `r_oem1`, etc.

  ❌ `wlan0` (Wi-Fi) is not considered vendor-owned. Vendor code can't modify it using these wrappers.

🔐 Why this matters

With Android's **Project Treble**, `/vendor` and `/system` are **independently updatable**. This wrapper ensures:

- Vendor code won't break after system OTA.
- Security is preserved (SELinux tightly controls what vendor can do).
- Framework updates don't affect low-level vendor network config.

🧠 Summary

Before Android 8	After Android 8+
Vendor calls `ip`, `iptables` directly	Vendor must use `ip-wrapper-1.0`, `iptables-wrapper-1.0`, etc.
No SELinux restrictions	SELinux must approve domain transition
Breakage risk on OTA	Safe and stable API to configure network
Full access to all commands	Only a whitelist of safe operations

---

If you want, I can show a full **example SELinux policy**, or how to configure a vendor HAL to set IPs using `ip-wrapper-1.0`.

25. explain in simple words its too confusing

Let's explain this in **very simple terms**, like you're new to Android internals:

🧠 The Problem

Before Android 8:

- Vendors (Qualcomm, Samsung, etc.) directly used system tools like:

  ```bash
  /system/bin/ip
  /system/bin/iptables
  ```

- These tools are part of the **system partition**, and they **change** when the phone gets updated (OTA).

🧨 So if the vendor's code expected an old version of the `ip` tool, but an update replaced it with a new one — **things could break**.

✅ The Solution: `netutils-wrapper-1.0`

Starting from Android 8:

> **Google created a middleman** tool called `netutils-wrapper-1.0` to act as a **safe and controlled gate** between vendor code and these system tools.

🔁 What's different now?

Old way (not allowed anymore):

```bash
/system/bin/ip addr add ...
```

New way (allowed):

```bash
/system/bin/ip-wrapper-1.0 addr add ...
```

💡 Why "wrapper"?

The wrapper:

- **Checks if the command is safe** (e.g. you're not changing Wi-Fi or system things).
- **Lets the command run** only if it follows strict rules.
- **Blocks** unsafe or system-only commands.

🛑 Example Rules (What vendors are allowed to do)

1. ✔️ You **can** add an IP to a modem interface like `rmnet_data0`
2. ❌ You **cannot** touch `wlan0` (that's Wi-Fi and belongs to Android framework)
3. ✔️ You **can** set up firewall for `rmnet_data0` using iptables
4. ❌ You **cannot** modify system-wide firewall rules

📂 What are these symlinks?

They are just **shortcuts** to the same file `netutils-wrapper-1.0`, for different tools:

Shortcut Command	Points to	Use For
`ip-wrapper-1.0`	`netutils-wrapper-1.0`	IP config
`iptables-wrapper-1.0`	`netutils-wrapper-1.0`	Firewall
`ndc-wrapper-1.0`	`netutils-wrapper-1.0`	Netd daemon
`tc-wrapper-1.0`	`netutils-wrapper-1.0`	Traffic control

🔒 SELinux Protection

SELinux enforces security.

- Vendor processes must **not** call `/system/bin/ip` or `/system/bin/netutils-wrapper-1.0` directly.
- They **must use the wrapper shortcuts**, like `/system/bin/ip-wrapper-1.0`
- Vendor SELinux policy must allow this access:

  ```c
  domain_auto_trans(vendor_process, netutils_wrapper_exec, netutils_wrapper)
  ```

🧠 Summary in 1 sentence:

> Android 8+ **forces vendors to use a safe middleman (netutils-wrapper)** to change network settings — only on their own hardware interfaces — and **blocks access to anything system-owned** like Wi-Fi or framework interfaces.

If you want a **real example** of a vendor adding IP using `ip-wrapper-1.0` or configuring a firewall using `iptables-wrapper-1.0`, I can show that too.

26. Model threading

bookmark_border
Methods marked as oneway don't block. For methods not marked as oneway, a client's method call blocks until the server has completed execution or called a synchronous callback (whichever comes first). Server method implementations can call at most one synchronous callback; extra callback calls are discarded and logged as errors. If a method is supposed to return values via callback and doesn't call its callback, this is logged as an error and reported as a transport error to the client.

Threads in passthrough mode
In passthrough mode, most calls are synchronous. However, to preserve the intended behavior that oneway calls don't block the client, a thread is created for each process. For details, see the HIDL overview.

Threads in binderized HALs
To serve incoming RPC calls (including asynchronous callbacks from HALs to HAL users) and death notifications, a threadpool is associated with each process that uses HIDL. If a single process implements multiple HIDL interfaces and/or death notification handlers, its threadpool is shared between all of them. When a process receives an incoming method call from a client, it picks a free thread from the threadpool and executes the call on that thread. If no free thread is available, it blocks until one is available.

If the server has only one thread, then calls into the server are completed in order. A server with more than one thread might complete calls out of order even if the client has only one thread. However, for a given interface object, oneway calls are guaranteed to be ordered (see Server threading model). For a multi-threaded server that hosts multiple interfaces, oneway calls to different interfaces might be processed concurrently with each other or other blocking calls.

Multiple nested calls are sent on the same hwbinder thread. For instance, if a process (A) makes a synchronous call from a hwbinder thread into process (B), and then process (B) makes a synchronous call back into process (A), the call is executed on the original hwbinder thread in (A) which is blocked on the original call. This optimization makes it possible to have a single threaded server able to handle nested calls, but it doesn't extend to cases where the calls travel through another sequence of IPC calls. For instance, if process (B) had made a binder/vndbinder call which called into a process (C) and then process (C) calls back into (A), it can't be served on the original thread in (A).

Server threading model
Except for passthrough mode, server implementations of HIDL interfaces live in a different process than the client and need one or more threads waiting for incoming method calls. These threads are the server's threadpool; the server can decide how many threads it wants running in its threadpool, and can use a threadpool size of one to serialize all calls on its interfaces. If the server has more than one thread in the threadpool, it can receive concurrent incoming calls on any of its interfaces (in C++, this means that shared data must be carefully locked).

One-way calls into the same interface are serialized. If a multi-threaded client calls method1 and method2 on interface IFoo, and method3 on interface IBar, method1 and method2 is always serialized, but method3 can run in parallel with method1 and method2.

A single client thread of execution can cause concurrent execution on a server with multiple threads in two ways:

oneway calls don't block. If a oneway call is executed and then a non-oneway is called, the server can execute the oneway call and the non-oneway call simultaneously.
Server methods that pass data back with synchronous callbacks can unblock the client as soon as the callback is called from the server.
For the second way, any code in the server function that executes after the callback is called can execute concurrently, with the server handling subsequent calls from the client. This includes code in the server function and automatic destructors that execute at the end of the function. If the server has more than one thread in its threadpool, concurrency issues arise even if calls are coming in from only one single client thread. (If any HAL served by a process needs multiple threads, all HALs have multiple threads because the threadpool is shared per-process.)

As soon as the server calls the provided callback, the transport can call the implemented callback on the client and unblock the client. The client proceeds in parallel with whatever the server implementation does after it calls the callback (which might include running destructors). Code in the server function after the callback is no longer blocking the client (as long as the server threadpool has enough threads to handle incoming calls), but might be executed concurrently with future calls from the client (unless the server threadpool has only one thread).

In addition to synchronous callbacks, oneway calls from a single-threaded client can be handled concurrently by a server with multiple threads in its threadpool, but only if those oneway calls are executed on different interfaces. oneway calls on the same interface are always serialized.

Note: We strongly encourage server functions to return as soon as they have called the callback function.

For example (in C++):

```
Return<void> someMethod(someMethod_cb _cb) {
    // Do some processing, then call callback with return data
```

```
    hidl_vec<uint32_t> vec = ...
    _cb(vec);
    // At this point, the client's callback is called,
    // and the client resumes execution.
    ...
    return Void(); // is basically a no-op
};
```

Client threading model
The threading model on the client differs between non-blocking calls (functions that are marked with the oneway keyword) and blocking calls (functions that don't have the oneway keyword specified).

Block calls
For blocking calls, the client blocks until one of the following happens:

Transport error occurs; the Return object contains an error state that can be retrieved with Return::isOk().
Server implementation calls the callback (if there was one).
Server implementation returns a value (if there was no callback parameter).
In case of success, the callback function the client passes as an argument is always called by the server before the function itself returns. The callback is executed on the same thread that the function call is made on, so implementers must be careful with holding locks during function calls (and avoid them altogether when possible). A function without a generates statement or a oneway keyword is still blocking; the client blocks until the server returns a Return<void> object.

One-way calls
When a function is marked oneway, the client returns immediately and doesn't wait for the server to complete its function call invocation. At the surface (and in aggregate), this means the function call takes half the time because it is executing half the code, but when writing implementations that are performance sensitive, this has some scheduling implications. Normally, using a one-way call causes the caller to continue to be scheduled whereas using a normal synchronous call causes the scheduler to immediately transfer from the caller to the callee process. This is a performance optimization in binder. For services where the one-way call must be executed in the target process with a high priority, the scheduling policy of the receiving service can be changed. In C++, using libhidltransport's method setMinSchedulerPolicy with the scheduler priorities and policies defined in sched.h ensures that all calls into the service run at least at the set scheduling policy and priority.

Let me explain **HIDL threading and oneway calls** in **simple words** with examples and diagrams so you understand:

🌐 What is the problem?

When an app or service calls a **HAL function** through HIDL, it happens through IPC (Inter-process communication). So we need to understand:

- What thread is executing the function?
- Is the caller blocked (waiting) or free to do other things?
- Can the server run multiple calls at the same time?

🧠 Key Concepts

1. Blocking Call (default)

A normal method (without `oneway`) **waits for the result**.

```cpp
Return<void> getTemperature(getTemperature_cb _cb);
```

- The **client is blocked** until the HAL responds.
- The **HAL (server)** executes the function and sends back a result via `_cb(...)`.

2. One-way Call (non-blocking)

A method marked with `oneway` doesn't return anything.

```cpp
oneway notifySensorChanged(int32_t value);
```

- The **client is not blocked**; it continues running.
- The **HAL (server)** will handle this call *later* in the background.

🧵 Server Threading

❓ Who executes the HAL method?

The server process (your HAL) has a **threadpool** that handles incoming calls.

- Example: If the threadpool has 4 threads, 4 different calls can be handled *at the same time*.
- If only 1 thread, calls are **serialized** (one by one).

⚠️ Concurrency Rule:

- All `oneway` **calls on the *same interface* are serialized**.
- But `oneway` calls on **different interfaces can be parallel.**

🧵 Client Threading

- If the method is **blocking** (no `oneway`), the **client waits** until it gets a response.
- If the method is **non-blocking** (`oneway`), the **client does not wait** — continues running.

🤝 Callback Execution

Let's say a method returns via a callback:

```cpp
Return<void> getData(getData_cb _cb) {
    // prepare data
    _cb(data);
    // you should return soon after callback
    return Void();
}
```

- The **moment you call** `_cb(...)`, the client is unblocked.
- Anything **after** `_cb(...)` may still be running on the server **while client proceeds**.

🔁 Nested Calls

Sometimes, the server may call back into the client:

- If both calls are in the **same binder thread**, it's allowed.
- But if there's a chain: A → B → C → A, it **won't use the original thread** in A, and may block or deadlock if threads are limited.

🚧 Passthrough Mode

In passthrough mode (used when HAL and framework are in the **same process**):

- `oneway` still behaves asynchronously.
- A new thread is created per process for `oneway` behavior.

🔧 Scheduling Control

If you want `oneway` **methods to run with high priority**, set scheduler policy:

```cpp
setMinSchedulerPolicy(policy, priority);
```

Example:

```cpp
setMinSchedulerPolicy(SCHED_FIFO, 5);
```

✅ Simple Summary Table

Type	Client Waits?	Server Threaded?	Ordered?
Blocking Call	✅ Yes	✅ Yes (if pool)	❌ Not ordered
oneway Call	❌ No	✅ Yes	✅ Ordered per interface

🧪 Example:

```cpp
// HAL interface
interface IExample {
    getData() generates(int32_t data);
    oneway notifyChange(int32_t code);
};
```

Client:

```cpp
example->getData([](int32_t data) {
   printf("Data is: %d\n", data);
}); // blocks until done

example->notifyChange(42); // returns immediately
```

Server:

```cpp
Return<void> IExample::getData(getData_cb _cb) {
    _cb(99);
    // client is unblocked now
    return Void();
}

Return<void> IExample::notifyChange(int32_t code) {
    LOG(INFO) << "Got notifyChange: " << code;
    // async, no return to client
    return Void();
}
```

Would you like a **diagram** showing the client-server flow with threadpool and oneway call examples?

> 27. explain callback example

Sure! Let's break down **callback in HIDL** in simple words, with a clear example.

🌟 What is a Callback in HIDL?

A **callback** is a way for the server (HAL) to **send data back** to the client (framework or app) **asynchronously**, **inside the same function call**.

✅ When do we use it?

When you want the client to request data from the HAL and get a result **during the same call**, you use a callback.

📦 Example HAL (HIDL interface)

```aidl
interface IMyHal {
    getSensorValue() generates(int32_t value);
};
```

This says:

> When the client calls **`getSensorValue()`**, the HAL will **generate** and return an **`int32_t`** value **via a callback**.

🧑‍💻 Client Code

Here's how the **client** (framework/app) calls this method:

```cpp
sp<IMyHal> myHal = IMyHal::getService();

myHal->getSensorValue([](int32_t value) {
    // This is the callback
    printf("Sensor value: %d\n", value);
});
```

➡️ What's happening?

- The client calls **`getSensorValue()`**.
- It passes a **lambda function** as the **callback**.
- The HAL executes the function and invokes the callback with a result.
- The lambda function prints the value.

🛠️ HAL Server Code (implementation)

```cpp
Return<void> MyHalImpl::getSensorValue(getSensorValue_cb _cb) {
    int32_t value = 123; // Imagine reading from a real sensor
    _cb(value);          // Call the callback to return value
    return Void();
}
```

➡️ What's happening?

- HAL gets the request.
- Reads sensor data (e.g., 123).

- Calls the **callback** `_cb(value)` to send result to the client.
- Returns **`Void()`** to finish the method.

⛓️ Flow Summary

```text
[Client]     ----calls--->     [HAL]
                  <---_cb()---
```

> 🔁 HAL does not directly return the value, but instead *calls the provided callback* to send it.

🔁 Multiple Callback Values (Advanced)

You can return multiple values using the callback:

HIDL:

```aidl
getStatus() generates(Status status, string message);
```

Client:

```cpp
myHal->getStatus([](Status s, const hidl_string& msg) {
    printf("Status: %d, Message: %s\n", (int)s, msg.c_str());
});
```

Server:

```cpp
Return<void> MyHalImpl::getStatus(getStatus_cb _cb) {
    _cb(Status::OK, "All systems go!");
    return Void();
}
```

🧠 Notes

- The **callback is always called by the HAL**.
- The **client waits (blocks)** until the callback is executed.
- If the HAL forgets to call the callback, it's a **transport error**.

Would you like a diagram of this flow or a full working example?
Great observation!

You're absolutely right to point that out. Let me clarify the confusion around the **`lambda`** and the **`generates(...)`** keyword in HIDL.

✅ What does `generates(...)` mean?

In HIDL, when you write:

```aidl
getSensorValue() generates(int32_t value);
```

It means:

> This method returns **asynchronously** using a **callback**. It does **not return the value directly**.

✅ Why we use a Lambda in C++?

When HIDL generates C++ code for this method, it **does not** return the `int32_t` value like a normal function.
Instead, it **expects you to pass a callback function** (which receives the result).

And in C++, the **lambda function** is a **short way** to define that callback.

🔧 C++ Binding Signature:

The actual method signature generated by HIDL for C++ looks like this:

```cpp
Return<void> getSensorValue(std::function<void(int32_t)> cb);
```

This means:

- The method returns `void`.
- You pass in a **function** (callback) that receives an `int32_t`.

That's why **we write**:

```cpp
myHal->getSensorValue([](int32_t value) {
    printf("Sensor value: %d\n", value);
});
```

The `[](...) { ... }` is a **lambda** — a compact way to define an inline function.

💡 If no `generates(...)`?

If the HIDL looked like:

```aidl
int32_t getSensorValue();
```

Then the C++ side would be a **direct return**:

```cpp
Return<int32_t> getSensorValue();
```

You would call it like:

```cpp
auto ret = myHal->getSensorValue();
if (ret.isOk()) {
    int32_t value = ret;
}
```

✅ Summary

HIDL Interface Style	C++ Call Style
`generates(...)`	Pass a **lambda function** as a callback
Regular return (`int32_t foo();`)	Call and get value via `Return<T>` directly

Let me know if you want a full server + client C++ example for `generates(...)`!

28. one vs generates

Great question!

In HIDL, the difference between `oneway` and `generates(...)` is all about **blocking vs non-blocking** and **how data is returned**.

Let's explain each in **simple terms with examples**.

✅ `generates(...)`:

Used for returning data via a callback

🔹 Meaning:

- The function is **blocking**.
- The client waits until the server calls the callback.
- The callback is used to return **values**.

🔹 HIDL:

```aidl
getSensorValue() generates(int32_t value);
```

🔹 C++ Client:

```cpp
myHal->getSensorValue([](int32_t value) {
    printf("Sensor value: %d\n", value);
});
```

🔹 Server:

```cpp
Return<void> MyHalImpl::getSensorValue(getSensorValue_cb _cb) {
    int32_t sensor = 123;
    _cb(sensor); // return value via callback
    return Void();
}
```

🔹 Use When:

You want the HAL to **return data to the client**.

✅ `oneway`:

Used for fire-and-forget calls (no return)

🔹 Meaning:

- The function is **non-blocking**.
- The client does **not wait** for the server.
- The function **cannot return data**.

🔹 HIDL:

```aidl
oneway notifyEvent(int32_t eventCode);
```

🔹 C++ Client:

```cpp
myHal->notifyEvent(101);  // Client doesn't wait
```

🔹 Server:

```cpp
Return<void> MyHalImpl::notifyEvent(int32_t eventCode) {
    // Just process the event
    return Void();
}
```

🔹 Use When:

- You **don’t need to return anything**.
- You want **high performance**, async behavior (e.g., notify, logging).

🆚 Comparison Table

Feature	`generates(...)`	`oneway`
Blocking?	✅ Yes (client waits)	❌ No (client doesn't wait)
Can return data?	✅ Yes (via callback)	❌ No
Uses callback?	✅ Yes	❌ No
Return type in C++	**`Return<void>(cb)`**	**`Return<void>(args...)`**
Use case	Get result from HAL	Notify HAL (fire and forget)

🧠 Tip:

- Use **`oneway`** for **commands** like **`start()`**, **`notify()`** where no response is needed.
- Use **`generates(...)`** for **queries** like **`getStatus()`**, **`getValue()`** where you need a result.

Would you like a combined example showing both in the same HAL?

29. Convert HAL modules

bookmark_border
You can update preexisting HAL modules to HIDL HAL modules by converting the header in hardware/libhardware/include/hardware.

Use c2hal
The c2hal tool handles most of the conversion work, reducing the number of required manual changes. For example, to generate a HIDL .hal file for the NFC HAL:

make c2hal
c2hal -r android.hardware:hardware/interfaces -randroid.hidl:system/libhidl/transport -p android.hardware.nfc@1.0 hardware/libhardware/include/hardware/nfc.h
These commands add files in hardware/interfaces/nfc/1.0/. Running hardware/interfaces/update-makefiles.sh from the $ANDROID_BUILD_TOP directory also adds the required makefile to the HAL. From here, you can make manual changes to fully convert the HAL.

c2hal activities
When you run c2hal, everything in the header file is transferred to .hal files.

c2hal identifies structs that contain function pointers in the provided header file and converts each struct into a separate interface file. For example, alloc_device_t is converted to the IAllocDevice HAL module (in the file IAllocDevice.hal).

All other data types are copied over into a types.hal file. Pound-defines are moved into enums, and items not a part of HIDL or not convertible (such as static-function declarations) are copied into comments marked with the text "NOTE".

Manual activities
The c2hal tool doesn't know what to do when it encounters certain constructs. For example, HIDL has no concept of raw pointers; because of this, when c2hal encounters a pointer in header files, it doesn't know whether the pointer should be interpreted as an array or as a reference to another object. Void pointers are also similarly opaque.

Field such as int reserved[7] must be manually removed during the transition to HIDL. Items such as the name of the return value should be updated to something more meaningful; for example, converting the return parameter of methods such as write in NFC from the autogenerated int32_t write_ret to Status status (where Status is a new enum containing possible NFC statuses).

Implement the HAL
After you have created .hal files to represent your HAL, you must generate the makefiles (Make or Soong) that create the language support in C++ and Java (unless the HAL uses a feature unsupported in Java). The ./hardware/interfaces/update-makefiles.sh script can automatically generate makefiles for HALs located in the hardware/interfaces directory (for HALs in other locations, simply update the script).

When the makefiles are up to date, you are ready to generate header files and implement methods. For details on implementing the generated interface, see HIDL C++ (for C++ implementations) or HIDL Java (for Java

Let’s break this down **step-by-step in simple words** with a real-world analogy and examples:

✅ Goal

You have an **old HAL written in C**, like `nfc.h`, and now you want to **convert it to HIDL** so it works with Android's modern binderized system.

🛠️ Step-by-Step Guide

🔹 1. What is `c2hal`?

Think of `c2hal` as a **translator** that takes your old C-style HAL header and **converts it into HIDL format** (`.hal` files).

🔹 2. Run the conversion tool

```sh
make c2hal
c2hal -r android.hardware:hardware/interfaces \
```

```
      -r android.hidl:system/libhidl/transport \
      -p android.hardware.nfc@1.0 \
      hardware/libhardware/include/hardware/nfc.h
```

What this does:

- Converts **`nfc.h`** into **`.hal`** files.
- Stores results in **`hardware/interfaces/nfc/1.0/`**.
- Creates files like:
  - **`INfc.hal`** (interface methods)
  - **`types.hal`** (structs, enums)

🔹 3. Run the Makefile generator

```sh
./hardware/interfaces/update-makefiles.sh
```

This auto-generates the required **Soong/Android.bp** build files.

🧰 What `c2hal` Converts Automatically

C/C++ Feature	What it becomes in HIDL
`struct` with function pointers	**`.hal`** interface (e.g., **`INfc`**)
`struct` without function pointers	HIDL **`struct`** in **`types.hal`**
`#define` constants	Converted into **`enum`** or **`const`**
Simple function return types	Auto-named like **`int32_t write_ret`**

🛑 Manual Fixes You Must Do

After **`c2hal`**, you **must manually clean up** the output:

❌ Remove things that don't belong in HIDL:

- Raw pointers (**`void*`**, **`uint8_t*`**) → use **`vec<uint8_t>`** or **`handle`**
- **`reserved[7]`** fields → delete
- Static/global functions → cannot be in HIDL, move to **`.cpp`** implementation
- Rename fields like **`write_ret`** → something meaningful like **`Status status`**

✏️ Example: Before and After

Original `nfc.h`:

```c
typedef struct {
    int (*write)(uint8_t* data, int len);
} nfc_device_t;
```

Auto-converted `INfc.hal`:

```hidl
interface INfc {
    write(vec<uint8_t> data, int32_t len) generates (int32_t write_ret);
};
```

Manual cleanup:

```hidl
enum Status : int32_t {
    OK,
    FAILED,
    TIMEOUT,
    // ...
};

interface INfc {
    write(vec<uint8_t> data) generates (Status status);
};
```

🔧 After Conversion: Implement the HAL

Now write your C++ implementation:

HIDL implementation (server-side):

```cpp
Return<void> Nfc::write(const hidl_vec<uint8_t>& data, write_cb _hidl_cb) {
    // send data to NFC chip...
    _hidl_cb(Status::OK);
    return Void();
}
```

🧠 Why This Is Important

By converting to HIDL:

- Your HAL becomes **stable**, **versioned**, and **binderized**
- You can **separate system/vendor** cleanly (important from Android 8+)
- Ensures compatibility with **Treble** architecture

Would you like a **real NFC `.hal` + implementation example** after conversion?

30. ↪
make c2hal c2hal -r android.hardware:hardware/interfaces \\ -r android.hidl:system/libhidl/transport \\ -p android.hardware.nfc@1.0 \\ hardware/libhardware/include/hardware/nfc.h
explain each option here

Let's break down the full **`c2hal`** command:

```sh
make c2hal
c2hal -r android.hardware:hardware/interfaces \
      -r android.hidl:system/libhidl/transport \
      -p android.hardware.nfc@1.0 \
      hardware/libhardware/include/hardware/nfc.h
```

✅ `make c2hal`

This builds the **`c2hal`** tool from source if it's not already built. It compiles the tool located in AOSP that performs the conversion of C headers to HIDL **`.hal`** files.

✅ `c2hal` command

This is the actual conversion command. Let's explain each option:

🔹 `-r android.hardware:hardware/interfaces`

Meaning:
This tells **`c2hal`** where to write the converted **`.hal`** files.

Explanation:

- The **`-r`** option maps a HIDL package name to a **file system directory**.
- Here, **`android.hardware`** HIDL packages are saved to **`hardware/interfaces`**.

🧠 Think of it like:

> "Put anything under **`android.hardware.*`** inside **`hardware/interfaces/`**."

🔹 `-r android.hidl:system/libhidl/transport`

Meaning:
Maps **`android.hidl`** (standard HIDL interfaces like **`IMemory`**, **`IBase`**, etc.) to their directory.

Explanation:

- Helps **`c2hal`** resolve references to standard HIDL types and interfaces.
- These are core HIDL interfaces shared across all HALs.

🔹 `-p android.hardware.nfc@1.0`

Meaning:
Specifies the target **package and version** for the generated HAL.

Explanation:

- **`-p`** stands for **package**.
- This will generate:
  - **`INfc.hal`**
  - **`types.hal`**

- Located in: `hardware/interfaces/nfc/1.0/`

🧠 Think of it like:

> "I'm generating the version 1.0 HIDL for NFC in the `android.hardware` namespace."

🔹 `hardware/libhardware/include/hardware/nfc.h`

Meaning:
This is the **input header file** — the old C HAL that you want to convert.

🧩 Summary Table

Option	Purpose
`make c2hal`	Builds the `c2hal` conversion tool
`-r android.hardware:hardware/interfaces`	Where to store converted HIDL for `android.hardware.*`
`-r android.hidl:system/libhidl/transport`	Where to find standard HIDL types like `IBase`
`-p android.hardware.nfc@1.0`	Target HIDL package and version
`hardware/libhardware/include/hardware/nfc.h`	Input: old-style HAL header to convert

Let me know if you'd like a **full walkthrough using another header** or how to handle the generated `.hal` files.

31. explain in simple about -r

Sure! Here's a **simple explanation of the `-r` option** in **`c2hal`**:

✅ What does `-r` mean?

`-r` stands for **"root mapping"** — it tells **`c2hal`**:

> "When you see this HIDL package name, put the files in this folder."

🧠 Think of it like:

> "**`android.hardware`** → put files in **`hardware/interfaces/`**"
> "**`android.hidl`** → put files in **`system/libhidl/transport/`**"

🔍 Example:

```bash
-r android.hardware:hardware/interfaces
```

Means:

- All HIDL files starting with **`android.hardware.*`** will go inside the **`hardware/interfaces`** folder.

So:

```hidl
android.hardware.nfc@1.0
```

➡️ will be stored in:

```swift
hardware/interfaces/nfc/1.0/
```

✅ Why do we need it?

Because **`c2hal`** needs to know:

- Where to find other HIDL types (like **`IBase`**, **`IMemory`**)
- And where to write the new **`.hal`** files

It maps **HIDL package names to real folder paths**.

Let me know if you'd like a visual or diagram to go with this!

Here's a **simple explanation** of the key concepts from the HIDL data types documentation you posted:

✅ HIDL Basics

HIDL (HAL Interface Definition Language) lets Android define HALs in a language-neutral way (works in both C++ and Java). It is designed to ensure stability and compatibility across system/vendor boundaries.

🧱 Basic Data Types

Type	Description
`int32_t`	32-bit signed integer (like C/C++)
`uint32_t`	32-bit unsigned integer
`float`	32-bit floating point number
`bool`	true or false
`string`	UTF-8 string internally; auto-converts to/from Java UTF-16
`handle`	Wraps native_handle_t for file descriptors

📦 Structs and Unions

- **Structs**: Group multiple variables together (like C). Example:

```
hidl

struct Point {
    int32_t x;
    int32_t y;
}
```

- **Unions**: Can store only one of the fields at a time. (No vec<string> etc. allowed inside.)

🔁 Arrays and Vectors

- **Sized array**: Fixed size.

```
hidl

int32_t[3] values;
```

- **Vector (`vec<T>`)**: Dynamic array.

```
hidl

vec<int32_t> dynamicArray;
```

- Nested vectors are allowed (like `vec<vec<int8_t>>`), but **not** `vec<vec<IFoo>>`.

⚠️ What's Not Allowed

- ❌ Anonymous structs or unions
- ❌ Pointers (*) — because HIDL is for IPC (cross-process), where raw pointers don't make sense.
- ❌ Self-referencing structs (no linked lists/trees)
- ❌ Nested interface vectors (e.g., `vec<vec<IFoo>>`)

🧩 Enums and Bitfields

- **Enum**: Like C++, with base type (e.g., `uint32_t`).

```
hidl

enum Color : uint32_t {
    RED = 0,
    GREEN = 1,
    BLUE = 2
};
```

- **bitfield<T>**: Used for flags (bitwise OR-ed enums).

```
hidl

typedef bitfield<Color> ColorFlags;
```

🧠 Interface Usage

- Interfaces are top-level services.
- Can only be passed:
  - As arguments or return types
  - As `vec<IFoo>`
- Can't be passed inside other types (like inside a struct or array)

🧵 Special Types

Type	Description
`memory`	Shared memory block (used with `IMemory`)
`hidl_handle`	Wraps a native_handle (FDs, used for shared resources like Ashmem)
`MQDescriptorSync`	Fast Message Queue (synchronized)
`MQDescriptorUnsync`	Fast Message Queue (unsynchronized)

📏 Java Differences

- No support for:
  - `union`
  - `handle` (Java can't handle file descriptors)
- `unsigned` types map to **signed** in Java (`uint32_t` → `int`)

📝 Annotations

Optional tags (like Java annotations) used by test tools (VTS), not by compiler. Example:

```hidl
@nullable
string name;
```

If you want a **diagram or cheatsheet**, I can create that too. Let me know.

32. Safe union

bookmark_border
safe_union in HIDL represents an explicitly tagged union type. This is similar to a union except safe_union keeps track of the underlying type and is compatible with Java. The safe_union type is available in Android 10 and higher for new and upgraded devices.

Syntax
A safe_union is expressed in HIDL exactly like a union or struct.

```
safe_union MySafeUnion {
    TypeA a;
    TypeB b;
    ...
};
```

Usage
At runtime, a safe_union is only ever one type. By default, it's the first type in the union. For instance, above, MySafeUnion is by default TypeA.

hidl-gen generates a custom class or struct for a safe_union in both C++ and Java. This class includes a discriminator for each member (in hidl_discriminator), a method to get the current discriminator (getDiscriminator), and setters and getters for each member. Each setter and getter is named exactly as its member. For instance, the getter for TypeA a is called "a", and it returns something of TypeA. The corresponding setter is also be called "a" and takes a parameter of TypeA. Setting the value in a safe_union updates the value of the discriminator as returned by getDiscriminator. Accessing a value from a discriminator that isn't the current discriminator aborts the program. For instance, if calling getDiscriminator on an instance of MySafeUnion returns hidl_discriminator::b, then trying to retrieve a aborts the program.

Monostate
A safe_union always has a value, but if it is desired to not have a value, use android.hidl.safe_union@1.0::Monostate as a placeholder. For instance, the following union can either be noinit (empty) or foo:

```
import android.hidl.safe_union@1.0::Monostate;

safe_union OptionalFoo {
    Monostate noinit;
    Foo foo;
};
```

Here's a **simple explanation of `safe_union` in HIDL**, with examples to make it easy to understand:

✅ What is `safe_union`?

`safe_union` is like a **normal union**, but **type-safe and Java-compatible**. It keeps track of **which field is currently active** and ensures **only that field is accessed**.

🚫 Problem with regular `union`

In a normal C-style **`union`**, you don't know which member currently holds a valid value. That's dangerous, especially across processes (IPC) or in Java.

✅ Why `safe_union` is better

- Tracks **which field is set** internally.
- Provides **getters and setters**.
- Accessing the **wrong field crashes** (better than silent bugs).
- Works **safely in Java and C++**.

🧱 Syntax Example

```hidl
safe_union MySafeUnion {
    int32_t num;
    string text;
};
```

🧠 How it works (C++/Java side):

Let's say we do this:

```cpp
MySafeUnion u;
u.text("hello");  // Sets the text and updates the discriminator
```

Now **`u.getDiscriminator()`** returns **`hidl_discriminator::text`**.

If you now try:

```cpp
int x = u.num();  // ❌ This crashes because current type is text
```

You **must only access the active field**.

🔄 Default Value

By default, the **first member** is active:

```hidl
safe_union MySafeUnion {
    int32_t num;  // default active
    string text;
};
```

🕳️ What if you want "no value"?

Use the **`Monostate`** type from HIDL:

```hidl
import android.hidl.safe_union@1.0::Monostate;

safe_union OptionalString {
    Monostate none;
    string value;
};
```

This works like a **`std::optional`** or **`nullable`** value:

- If set to **`none`**, then it's "empty"
- If set to **`value`**, then it holds a string

🧪 Summary

Feature	`union`	`safe_union`
Keeps track of active field	❌ No	✅ Yes
Crashes on invalid access	❌ Undefined	✅ Yes
Java support	❌ No	✅ Yes
Can have "no value"	❌ No	✅ Yes (via **`Monostate`**)

Would you like a C++ or Java example using this?

33. explain in simple words

Sure! Let me explain **`safe_union`** in **very simple words**:

🔄 What is a union?

Normally, in C/C++, a **`union`** is a container that can hold **one value at a time**, but it can be **of different types**.
For example:

```c
union Example {
    int a;
    float b;
};
```

You can store **`a` or `b`**, but **not both at the same time**.

❌ Problem with normal `union`

In a normal union:

- It **doesn't remember** what type you stored.
- If you store a float and later read it as an int — 💥 **bad things can happen** (crash or garbage).

✅ What is `safe_union` in HIDL?

`safe_union` is a **safer version** of union that:

- **Remembers** what type of value you stored.
- **Prevents wrong access**.
- Works well with **Java and C++** (for Android HALs).

📦 Example:

```hidl
safe_union MyData {
    int32_t number;
    string name;
};
```

Now when using this:

- You can **store either number or name**.
- If you store **`name`**, and someone tries to read **`number`**, the program will crash (because that's not the current type).

🔍 How it works:

- It has an internal tracker called **discriminator** — like a tag to know which value is active.
- You use **setters and getters** like **`setName()`** or **`getNumber()`**.
- It checks: "Is **`number`** active?" before giving it to you.

🕳️ What if I want it to be empty?

Use this:

```hidl
hidl

import android.hidl.safe_union@1.0::Monostate;

safe_union OptionalName {
    Monostate none;
    string name;
};
```

Now it’s like saying:

- Either the value is **nothing** (`none`),
- Or it's a `name`.

🧠 Summary:

Feature	Normal `union`	`safe_union`
Keeps track of value	❌ No	✅ Yes
Prevents wrong access	❌ No	✅ Yes (crash if wrong)
Supports Java	❌ No	✅ Yes
Can be "empty"	❌ No	✅ Yes (Monostate)

Would you like to see how it's used in C++ or HIDL code too?

34. Interface versioning

bookmark_border
HIDL requires every interface written in HIDL be versioned. After a HAL interface is published, it is frozen and any further changes must be made to a new version of that interface. While a given published interface can't be modified, it can be extended by another interface.

HIDL code structure
HIDL code is organized in user-defined types, interfaces, and packages:

User-defined types (UDTs). HIDL provides access to a set of primitive data types that can be used to compose more complex types via structures, unions, and enumerations. UDTs are passed to methods of interfaces, and can be defined at the level of a package (common to all interfaces) or locally to an interface.
Interfaces. As a basic building block of HIDL, an interface consists of UDT and method declarations. Interfaces can also inherit from another interface.
Packages. Organizes related HIDL interfaces and the data types on which they operate. A package is identified by a name and a version and includes the following:
Data-type definition file called types.hal.
Zero or more interfaces, each in their own .hal file.
The data-type definition file types.hal contains only UDTs (all package-level UDTs are kept in a single file). Representations in the target language are available to all interfaces in the package.

Versioning philosophy
A HIDL package (such as android.hardware.nfc), after being published for a given version (such as 1.0), is immutable; it cannot be changed. Modifications to the interfaces in the package or any changes to its UDTs can take place only in another package.

In HIDL, versioning applies at the package level, not at the interface level, and all interfaces and UDTs in a package share the same version. Package versions follow semantic versioning without the patch level and build-metadata components. Within a given package, a minor version bump implies the new version of the package is backwards-compatible with the old package and a major version bump implies the new version of the package isn't backwards-compatible with the old package.

Conceptually, a package can relate to another package in one of several ways:

Not at all.
Package-level backwards-compatible extensibility. This occurs for new minor-version uprevs (next incremented revision) of a package; the new package has the same name and major version as the old package, but a higher minor version. Functionally, the new package is a superset of the old package, meaning:
Top-level interfaces of the parent package are present in the new package, though the interfaces might have new methods, new interface-local UDTs (the interface-level extension described below), and new UDTs in types.hal.
New interfaces can also be added to the new package.
All data types of the parent package are present in the new package and can be handled by the (possibly reimplemented) methods from the old package.
New data types can also be added for use by either new methods of uprev'ed existing interfaces, or by new interfaces.
Interface-level backwards-compatible extensibility. The new package can also extend the original package by consisting of logically separate interfaces that simply provide additional functionality, and not the core one. For this purpose, the following might be desirable:
Interfaces in the new package need recourse to the data types of the old package.
Interfaces in new package can extend interfaces of one or more old packages.
Extend the original backwards-incompatibility. This is a major-version uprev of the package and there need not be any correlation between the two. To the extent that there is, it can be expressed with a combination of types from the older version of the package, and inheritance of a subset of old-package interfaces.
Interface structuring
For a well structured interface, adding new types of functionality that aren't part of the original design should require a modification to the HIDL interface. Conversely, if you can or expect to make a change on both sides of the interface that introduces new functionality without changing the interface itself, then the interface isn't structured.

Treble supports separately compiled vendor and system components in which the vendor.img on a device and the system.img can be compiled separately. All interactions between vendor.img and system.img must be explicitly and thoroughly defined so they can continue to work for many years. This includes many API surfaces, but a major surface is the IPC mechanism HIDL uses for interprocess communication on the system.img/vendor.img boundary.

Requirements
All data passed through HIDL must be explicitly defined. To ensure an implementation and client can continue to work together even when compiled separately or developed on independently, data must adhere to the following requirements:

Can be described in HIDL directly (using structs enums, etc.) with semantic names and meaning.
Can be described by a public standard such as ISO/IEC 7816.
Can be described by a hardware standard or physical layout of hardware.
Can be opaque data (such as public keys, ids, etc.) if necessary.
If opaque data is used, it must be read only by one side of the HIDL interface. For example, if vendor.img code gives a component on the system.img a string message or vec<uint8_t> data, that data cannot be parsed by the system.img itself; it can only be passed back to vendor.img to interpret. When passing a value from vendor.img to vendor code on system.img or to another device, the format of the data and how it is to be interpreted must be exactly described and is still part of the interface.

Guidelines
You should be able to write an implementation or client of a HAL using only the .hal files (i.e. you shouldn't need to look at the Android source or public standards). We recommend specifying the exact required behavior. Statements such as "an implementation might do A or B" encourage implementations to become intertwined with the clients they are developed with.

HIDL code layout
HIDL includes core and vendor packages.

Core HIDL interfaces are those specified by Google. The packages they belong to start with android.hardware. and are named by subsystem, potentially with nested levels of naming. For example, the NFC package is named android.hardware.nfc and the camera package is android.hardware.camera. In general, a core package has the name android.hardware.[name1].[name2]…. HIDL packages have a version in addition to their name. For example, the package android.hardware.camera might be at version 3.4; this is important, as the version of a package affects its placement in the source tree.

All core packages are placed under hardware/interfaces/ in the build system. The package android.hardware.[name1].[name2]… at version $m.$n is under hardware/interfaces/name1/name2/…/$m.$n/; package android.hardware.camera version 3.4 is in directory hardware/interfaces/camera/3.4/. A hard-coded mapping exists between the package prefix android.hardware. and the path hardware/interfaces/.

Non-core (vendor) packages are those produced by the SoC vendor or ODM. The prefix for non-core packages is vendor.$(VENDOR).hardware. where $(VENDOR)refers to an SoC vendor or OEM/ODM. This maps to the path vendor/$(VENDOR)/interfaces in the tree (this mapping is also hard-coded).

Fully qualified user-defined-type names
In HIDL, every UDT has a fully qualified name that consists of the UDT name, the package name where the UDT is defined, and the package version. The fully qualified name is used only when instances of the type are declared and not where the type itself is defined. For example, assume package android.hardware.nfc, version 1.0 defines a struct named NfcData. At the site of the declaration (whether in types.hal or within an interface's declaration), the declaration simply states:

```
struct NfcData {
    vec<uint8_t> data;
};
```

When declaring an instance of this type (whether within a data structure or as a method parameter), use the fully qualified type name:

```
android.hardware.nfc@1.0::NfcData
```

The general syntax is PACKAGE@VERSION::UDT, where:

PACKAGE is the dot-separated name of a HIDL package (e.g., android.hardware.nfc).
VERSION is the dot-separated major.minor-version format of the package (e.g., 1.0).
UDT is the dot-separated name of a HIDL UDT. Since HIDL supports nested UDTs and HIDL interfaces can contain UDTs (a type of nested declaration), dots are used to access the names.
For example, if the following nested declaration was defined in the common types file in package android.hardware.example version 1.0:

```
// types.hal
package android.hardware.example@1.0;
struct Foo {
    struct Bar {
        // ...
    };
    Bar cheers;
};
```

The fully qualified name for Bar is android.hardware.example@1.0::Foo.Bar. If, in addition to being in the above package, the nested declaration were in an interface called IQuux:

```
// IQuux.hal
package android.hardware.example@1.0;
interface IQuux {
   struct Foo {
      struct Bar {
         // ...
      };
      Bar cheers;
   };
   doSomething(Foo f) generates (Foo.Bar fb);
};
```

The fully qualified name for Bar is android.hardware.example@1.0::IQuux.Foo.Bar.

In both cases, Bar can be referred to as Bar only within the scope of the declaration of Foo. At the package or interface level, you must refer to Bar via Foo: Foo.Bar, as in the declaration of method doSomething above. Alternatively, you could declare the method more verbosely as:

```
// IQuux.hal
doSomething(android.hardware.example@1.0::IQuux.Foo f) generates (android.hardware.example@1.0::IQuux.Foo.Bar fb);
```

Fully qualified enumeration values

If a UDT is an enum type, then each value of the enum type has a fully qualified name that starts with the fully qualified name of the enum type, followed by a colon, then followed by the name of the enum value. For example, assume package android.hardware.nfc, version 1.0 defines an enum type NfcStatus:

```
enum NfcStatus {
   STATUS_OK,
   STATUS_FAILED
};
```

When referring to STATUS_OK, the fully qualified name is:

```
android.hardware.nfc@1.0::NfcStatus:STATUS_OK
```

The general syntax is PACKAGE@VERSION::UDT:VALUE, where:

PACKAGE@VERSION::UDT is the exact same fully qualified name for the enum type.
VALUE is the value's name.

Auto-inference rules

A fully qualified UDT name doesn't need to be specified. A UDT name can safely omit the following:

The package, e.g. @1.0::IFoo.Type
Both package and version, e.g. IFoo.Type
Caution: UDT names missing a version but specifying a package present aren't allowed.
HIDL attempts to complete the name using auto-interference rules (lower rule number means higher priority).

Rule 1

If no package and version is provided, a local name lookup is attempted. Example:

```
interface Nfc {
   typedef string NfcErrorMessage;
   send(NfcData d) generates (@1.0::NfcStatus s, NfcErrorMessage m);
};
```

NfcErrorMessage is looked up locally, and the typedef above it is found. NfcData is also looked up locally, but as it is not defined locally, rule 2 and 3 are used. @1.0::NfcStatus provides a version, so rule 1 doesn't apply.

Rule 2

If rule 1 fails and a component of the fully qualified name is missing (package, version, or package and version), the component is autofilled with information from the current package. The HIDL compiler then looks in the current file (and all imports) to find the autofilled fully qualified name. Using the example above, assume the declaration of ExtendedNfcData was made in the same package (android.hardware.nfc) at the same version (1.0) as NfcData, as follows:

```
struct ExtendedNfcData {
   NfcData base;
```

```
    // ... additional members
};
```

The HIDL compiler fills out the package name and version name from the current package to produce the fully qualified UDT name android.hardware.nfc@1.0::NfcData. As the name exists in the current package (assuming it is imported properly), it is used for the declaration.

A name in the current package is imported only if one of the following is true:

It is imported explicitly with an import statement.
It is defined in types.hal in the current package
The same process is followed if NfcData was qualified by only the version number:

```
struct ExtendedNfcData {
    // autofill the current package name (android.hardware.nfc)
    @1.0::NfcData base;
    // ... additional members
};
```

Rule 3
If rule 2 fails to produce a match (the UDT isn't defined in the current package), the HIDL compiler scans for a match within all imported packages. Using the above example, assume ExtendedNfcData is declared in version 1.1 of package android.hardware.nfc, 1.1 imports 1.0 as it should (see Package-Level Extensions), and the definition specifies only the UDT name:

```
struct ExtendedNfcData {
    NfcData base;
    // ... additional members
};
```

The compiler looks for any UDT named NfcData and finds one in android.hardware.nfc at version 1.0, resulting in a fully qualified UDT of android.hardware.nfc@1.0::NfcData. If more than one match is found for a given partially qualified UDT, the HIDL compiler throws an error.

Example
Using rule 2, an imported type defined in the current package is favored over an imported type from another package:

```
// hardware/interfaces/foo/1.0/types.hal
package android.hardware.foo@1.0;
struct S {};

// hardware/interfaces/foo/1.0/IFooCallback.hal
package android.hardware.foo@1.0;
interface IFooCallback {};

// hardware/interfaces/bar/1.0/types.hal
package android.hardware.bar@1.0;
typedef string S;

// hardware/interfaces/bar/1.0/IFooCallback.hal
package android.hardware.bar@1.0;
interface IFooCallback {};

// hardware/interfaces/bar/1.0/IBar.hal
package android.hardware.bar@1.0;
import android.hardware.foo@1.0;
interface IBar {
    baz1(S s); // android.hardware.bar@1.0::S
    baz2(IFooCallback s); // android.hardware.foo@1.0::IFooCallback
};
```

S is interpolated as android.hardware.bar@1.0::S, and is found in bar/1.0/types.hal (because types.hal is automatically imported).
IFooCallback is interpolated as android.hardware.bar@1.0::IFooCallback using rule 2, but it cannot be found because bar/1.0/IFooCallback.hal isn't imported automatically (as types.hal is). Thus, rule 3 resolves it to android.hardware.foo@1.0::IFooCallback instead, which is imported via import android.hardware.foo@1.0;).
types.hal
Every HIDL package contains a types.hal file containing UDTs that are shared among all interfaces participating in that package. HIDL types are always public; regardless of whether a UDT is declared in types.hal or within an interface declaration, these types are accessible outside of the scope where they are defined. types.hal isn't meant to describe the public API of a package, but rather to host UDTs used by all interfaces within the package. Due to the nature of HIDL, all UDTs are a part of the interface.

types.hal consists of UDTs and import statements. Because types.hal is made available to every interface of the package (it is an implicit import), these import statements are package-level by definition. UDTs in types.hal can also incorporate UDTs and interfaces thus imported.

For example, for an IFoo.hal:

```
package android.hardware.foo@1.0;
// whole package import
import android.hardware.bar@1.0;
// types only import
import android.hardware.baz@1.0::types;
// partial imports
import android.hardware.qux@1.0::IQux.Quux;
// partial imports
import android.hardware.quuz@1.0::Quuz;
```

The following are imported:

android.hidl.base@1.0::IBase (implicitly)
android.hardware.foo@1.0::types (implicitly)
Everything in android.hardware.bar@1.0 (including all interfaces and its types.hal)
types.hal from android.hardware.baz@1.0::types (interfaces in android.hardware.baz@1.0 aren't imported)
IQux.hal and types.hal from android.hardware.qux@1.0
Quuz from android.hardware.quuz@1.0 (assuming Quuz is defined in types.hal, the entire types.hal file is parsed, but types other than Quuz aren't imported).
Interface-level versioning
Each interface within a package resides in its own file. The package the interface belongs to is declared at the top of the interface using the package statement. Following the package declaration, zero or more interface-level imports (partial or whole-package) might be listed. For example:

```
package android.hardware.nfc@1.0;
```

In HIDL, interfaces can inherit from other interfaces using the extends keyword. For an interface to extend another interface, it must have access to it via an import statement. The name of the interface being extended (the base interface) follows the rules for type-name qualification explained above. An interface can inherit only from one interface; HIDL doesn't support multiple inheritance.

The uprev versioning examples below use the following package:

```
// types.hal
package android.hardware.example@1.0
struct Foo {
    struct Bar {
        vec<uint32_t> val;
    };
};

// IQuux.hal
package android.hardware.example@1.0
interface IQuux {
    fromFooToBar(Foo f) generates (Foo.Bar b);
}
```

Uprev rules
To define a package package@major.minor, either A or all of B must be true:

Rule A "Is a start minor version": All previous minor versions, package@major.0, package@major.1, …, package@major.(minor-1) must not be defined.
OR
Rule B
All of the following is true:

"Previous minor version is valid": package@major.(minor-1) must be defined and follow the same rule A (none of package@major.0 through package@major.(minor-2) are defined) or rule B (if it is an uprev from @major.(minor-2));

AND

"Inherit at least one interface with the same name": There exists an interface package@major.minor::IFoo that extends package@major.(minor-1)::IFoo (if the previous package has an interface);

AND

"No inherited interface with a different name": There must not exist package@major.minor::IBar that extends package@major.(minor-1)::IBaz, where IBar and IBaz are two different names. If there is an interface with the same name, package@major.minor::IBar must extend package@major.(minor-k)::IBar such that no IBar exists with a smaller k.
Because of rule A:

The package can start with any minor version number (for example, android.hardware.biometrics.fingerprint starts at @2.1.)
The requirement "android.hardware.foo@1.0 isn't defined" means the directory hardware/interfaces/foo/1.0 shouldn't even exist.
However, rule A doesn't affect a package with the same package name but a different major version (for example, android.hardware.camera.device has both @1.0 and @3.2 defined; @3.2 doesn't need to interact with @1.0.) Hence, @3.2::IExtFoo can extend @1.0::IFoo.

Provided the package name is different, package@major.minor::IBar can extend from an interface with a different name (for example, android.hardware.bar@1.0::IBar can extend android.hardware.baz@2.2::IBaz). If an interface doesn't explicitly declare a super type with the extend keyword, it extends android.hidl.base@1.0::IBase (except IBase itself).

B.2 and B.3 must be followed at the same time. For example, even if android.hardware.foo@1.1::IFoo extends android.hardware.foo@1.0::IFoo to pass rule B.2, if an android.hardware.foo@1.1::IExtBar extends android.hardware.foo@1.0::IBar, this is still not a valid uprev.

Uprev interfaces
To uprev android.hardware.example@1.0 (defined above) to @1.1:

```
// types.hal
package android.hardware.example@1.1;
import android.hardware.example@1.0;

// IQuux.hal
package android.hardware.example@1.1
interface IQuux extends @1.0::IQuux {
    fromBarToFoo(Foo.Bar b) generates (Foo f);
}
```

This is a package-level import of version 1.0 of android.hardware.example in types.hal. While no new UDTs are added in version 1.1 of the package, references to UDTs in version 1.0 are still needed, hence the package-level import in types.hal. (The same effect could have been achieved with an interface-level import in IQuux.hal.)

In extends @1.0::IQuux in the declaration of IQuux, we specified the version of IQuux that is being inherited (disambiguation is required because IQuux is used to declare an interface and to inherit from an interface). As declarations are simply names that inherit all package and version attributes at the site of the declaration, the disambiguation must be in the name of the base interface; we could have used the fully qualified UDT as well, but that would have been redundant.

The new interface IQuux doesn't re-declare method fromFooToBar() it inherits from @1.0::IQuux; it simply lists the new method it adds fromBarToFoo(). In HIDL, inherited methods can not be declared again in the child interfaces, so the IQuux interface cannot declare the fromFooToBar() method explicitly.

Key Point: In HIDL, every inherited method from a base class must be explicitly implemented in the inheriting class. If a method implementation needs to fall back to the method implementation of the base class, the fallback must be in the implementation.
Uprev conventions
Sometimes interface names must rename the extending interface. We recommend that enum extensions, structs, and unions have the same name as what they extend unless they are sufficiently different to warrant a new name. Examples:

```
// in parent hal file
enum Brightness : uint32_t { NONE, WHITE };

// in child hal file extending the existing set with additional similar values
enum Brightness : @1.0::Brightness { AUTOMATIC };

// extending the existing set with values that require a new, more descriptive name:
enum Color : @1.0::Brightness { HW_GREEN, RAINBOW };
```

If a method can have a new semantic name (for instance fooWithLocation) then that is preferred. Otherwise, it should be named similarly to what it is extending. For example, the method foo_1_1 in @1.1::IFoo can replace the functionality of the foo method in @1.0::IFoo if there is no better alternative name.

Package-level versioning
HIDL versioning occurs at the package level; after a package is published, it is immutable (its set of interfaces and UDTs cannot be changed). Packages can relate to each other in several ways, all of which are expressible via a combination of interface-level inheritance and building of UDTs by composition.

However, one type of relationship is strictly defined and must be enforced: Package-level backwards-compatible inheritance. In this scenario, the parent package is the package being inherited from and the child package is the one extending the parent. Package-level backwards-compatible inheritance rules are as follows:

All top-level interfaces of the parent package are inherited from by interfaces in the child package.
New interfaces can also be added the new package (no restrictions about relationships to other interfaces in other packages).
New data types can also be added for use by either new methods of uprev'ed existing interfaces, or by new interfaces.
These rules can be implemented using HIDL interface-level inheritance and UDT composition, but require meta-level knowledge to know these relationships constitute a backwards-compatible package extension. This knowledge is inferred as follows:

Key Point: For package package at version major.minor, if a package exists at major.(minor-1), then package@major.minor is a minor uprev, and must follow the rules for backwards-compatibility.
If a package meets this requirement, hidl-gen enforces backwards-compatibility rules.

Here's a **simple explanation** of the whole "HIDL Interface Versioning" system, broken down for clarity:

🧱 What is HIDL?

HIDL (HAL Interface Definition Language) defines how Android system and vendor (hardware-specific) code talk to each other. It's like a contract between the Android system and hardware drivers.

🗃️ Why Versioning is Important?

Once a HAL (e.g., NFC, Camera) is defined and released, it **must not be changed**. Instead, if new features are added, a **new version** is created. This makes it safe for devices to upgrade system software without breaking vendor HALs.

📦 HIDL Package Structure

A HIDL package includes:

- **`types.hal`**: shared data structures.
- One or more interface files like **`IFoo.hal`**, **`IBar.hal`**.

Example:

```plaintext
hardware/interfaces/camera/3.4/
├── types.hal
├── ICameraDevice.hal
```

Package name: **`android.hardware.camera@3.4`**

🔢 Versioning Rules

- **Versioning is done at the package level**, not individual interfaces.
- Once **`android.hardware.foo@1.0`** is released, it **cannot be changed**.
- To add features, create a new version like **`@1.1`**.

🧬 Extending Interfaces

You can create a new interface version that **extends** the old one:

```hidl
// ICameraDevice.hal in @3.4
interface ICameraDevice {
    open();
}

// ICameraDevice.hal in @3.5
interface ICameraDevice extends @3.4::ICameraDevice {
    openAdvanced();
}
```

- You **cannot redefine** old methods.
- You can add new ones.
- The system will know how to fall back to old methods on older devices.

🧠 Fully Qualified Names (FQNs)

HIDL needs to know exactly **which version** of a type or interface you mean.

Example:

```hidl
android.hardware.camera@3.4::CameraMetadata
```

General pattern:

```plaintext
PACKAGE@VERSION::Thing
```

🔄 Auto-Inference of Names

If you write just `CameraMetadata`, HIDL will:

1. Check in the current file.
2. Then in the current package's `types.hal`.
3. Then in any imported packages.

🛑 Common Restrictions

- You **must not mix interface names** when uprevving.
  - `IBar` in 1.1 cannot extend `IBaz` in 1.0.
- All interfaces in the new version must **extend same-named interfaces** from the previous version.
- You **can't have** `@1.1::IBar` without `@1.0::IBar`.

🔲 Package Versioning vs Interface Inheritance

- `package@1.1::IFoo` must extend `package@1.0::IFoo` if that interface existed before.
- New interfaces (`INewThing`) can be added freely.
- You can import UDTs and use them across versions.

✅ Good Practices

- Keep types shared by all interfaces in `types.hal`.
- Use meaningful names.
- Prefer adding new methods instead of changing existing ones.
- Use enums/structs/typed data—no raw pointers or void*.

Would you like a **diagram**, **real example**, or simplified **file layout** for a typical uprev scenario (e.g., `IFoo@1.0` → `IFoo@1.1`)?

35. ↪
Keep types shared by all interfaces in `types.hal`.
explain more about this in simple words

Sure! Let's break it down simply:

✅ What does "Keep types shared by all interfaces in `types.hal`" mean?

When you're writing a HIDL HAL, you'll define:

- **Interfaces** like **`ICameraDevice`**, **`ISensor`**
- **Data structures** (types) like **`CameraSettings`**, **`SensorData`**

Now, if **many interfaces need to use the same types**, instead of writing those types again and again in each interface, you **put them in a common place** called **`types.hal`**.

🔍 Why use `types.hal`?

Because:

- It avoids **duplication**.
- It keeps the code **organized**.
- All interfaces in the same package can **automatically use those types**.

🔧 Example:

Let's say you have this type:

```hidl
// This is a common data type
struct Resolution {
    uint32_t width;
    uint32_t height;
}
```

If you have two interfaces:

- **`ICamera`** → uses **`Resolution`**
- **`IDisplay`** → also uses **`Resolution`**

📦 Instead of writing **`Resolution`** twice in both interfaces, you put it once in **`types.hal`**:

```hidl
// types.hal
package android.hardware.display@1.0;

struct Resolution {
    uint32_t width;
    uint32_t height;
}
```

Then in your interfaces:

```hidl
// ICamera.hal
import android.hardware.display@1.0::types;

interface ICamera {
    setResolution(Resolution res);
}

// IDisplay.hal
```

```
import android.hardware.display@1.0::types;

interface IDisplay {
    applyResolution(Resolution r);
}
```

🧠 Summary:

- **`types.hal`** is like a **common toolbox** for shared data types.
- Use it to **store structs, enums, typedefs** that are **used by multiple interfaces**.
- This makes the code **cleaner, easier to maintain**, and **less error-prone**.

Would you like a real file structure example or want me to write a sample **`types.hal`** + **`IService.hal`** pair?

36. Code style guide

bookmark_border
The HIDL code style resembles C++ code in the Android framework, with 4-space indents and mixed-case filenames. Package declarations, imports, and docstrings are similar to those in Java, with slight modifications.

The following examples for IFoo.hal and types.hal illustrate HIDL code styles and provide quick links to details on each style (IFooClientCallback.hal, IBar.hal, and IBaz.hal have been omitted).

hardware/interfaces/foo/1.0/IFoo.hal

```
/*
 * (License Notice)
 */

package android.hardware.foo@1.0;

import android.hardware.bar@1.0::IBar;

import IBaz;
import IFooClientCallback;

/**
 * IFoo is an interface that…
 */
interface IFoo {

    /**
     * This is a multiline docstring.
     *
     * @return result 0 if successful, nonzero otherwise.
     */
     foo() generates (FooStatus result);

    /**
     * Restart controller by power cycle.
     *
     * @param bar callback interface that…
     * @return result 0 if successful, nonzero otherwise.
     */
    powerCycle(IBar bar) generates (FooStatus result);

    /** Single line docstring. */
    baz();


    /**
     * The bar function.
     *
     * @param clientCallback callback after function is called
     * @param baz related baz object
     * @param data input data blob
     */
    bar(IFooClientCallback clientCallback,
        IBaz baz,
        FooData data);

};
```

hardware/interfaces/foo/1.0/types.hal

```
/*
 * (License Notice)
 */
```

```
package android.hardware.foo@1.0;

/** Replied status. */
enum Status : int32_t {
   OK,
   /* invalid arguments */
   ERR_ARG,
   /* note, no transport related errors */
   ERR_UNKNOWN = -1,
};

struct ArgData {
   int32_t[20]  someArray;
   vec<uint8_t> data;
};
```

Naming conventions
Function names, variable names, and filenames should be descriptive; avoid over-abbreviation. Treat acronyms as words (for example, use INfc instead of INFC).

Directory structure and file naming
The directory structure should appear as follows:

ROOT-DIRECTORY
MODULE
SUBMODULE (optional, could be more than one level)
VERSION
Android.mk
IINTERFACE_1.hal
IINTERFACE_2.hal
...
IINTERFACE_N.hal
types.hal (optional)
Where:

ROOT-DIRECTORY is:
hardware/interfaces for core HIDL packages.
vendor/VENDOR/interfaces for vendor packages, where VENDOR refers to an SoC vendor or an OEM/ODM.
MODULE should be one lowercase word that describes the subsystem (for example, nfc). If more than one word is needed, use nested SUBMODULE. There can be more than one level of nesting.
VERSION should be the exact same version (major.minor) as described in Versions.
IINTERFACE_X should be the interface name with UpperCamelCase/PascalCase (for example, INfc) as described in Interface names.
Example:

hardware/interfaces
nfc
1.0
Android.mk
INfc.hal
INfcClientCallback.hal
types.hal
Note: All files must have non-executable permissions (in Git).

Package names
Package names must use the following fully-qualified name (FQN) format (referred to as PACKAGE-NAME):

PACKAGE.MODULE[.SUBMODULE[.SUBMODULE[...]]]@VERSION
Where:

PACKAGE is the package that maps to the ROOT-DIRECTORY. In particular, PACKAGE is:
android.hardware for core HIDL packages (mapping to hardware/interfaces).
vendor.VENDOR.hardware for vendor packages, where VENDOR refers to an SoC vendor or an OEM/ODM (mapping to vendor/VENDOR/interfaces).
MODULE[.SUBMODULE[.SUBMODULE[...]]]@VERSION are the exact same folder names in the structure described in Directory structure.
Package names should be lowercase. If they are more than one word long, the words should either be used as submodules or written in

snake_case.
No spaces are allowed.
The FQN is always used in package declarations.

Versions
Versions should have the following format:

MAJOR.MINOR
Both the MAJOR and the MINOR version should be a single integer. HIDL uses semantic versioning rules.

Imports
An import has one of the following three formats:

Whole-package imports: import PACKAGE-NAME;
Partial imports: import PACKAGE-NAME::UDT; (or, if the imported type is in the same package,import UDT;
Types-only imports: import PACKAGE-NAME::types;
The PACKAGE-NAME follows the format in Package names. The current package's types.hal (if it exists) is automatically imported (do not import it explicitly).

Fully qualified names (FQNs)
Use fully qualified names for a user-defined type import only when necessary. Omit PACKAGE-NAME if the import type is in the same package. An FQN must not contain spaces. Example of a fully qualified name:

android.hardware.nfc@1.0::INfcClientCallback
In another file under android.hardware.nfc@1.0, refer to the above interface as INfcClientCallback. Otherwise, use only the fully qualified name.

Grouping and ordering imports
Use an empty line after package declaration (before the imports). Each import should occupy a single line and should not be indented. Group imports in the following order:

Other android.hardware packages (use fully qualified names).
Other vendor.VENDOR packages (use fully qualified names).
Each vendor should be a group.
Order vendors alphabetically.
Imports from other interfaces in the same package (use simple names).
Use an empty line between groups. Inside each group, sort imports alphabetically. Example:

import android.hardware.nfc@1.0::INfc;
import android.hardware.nfc@1.0::INfcClientCallback;

/* Importing the whole module. */
import vendor.barvendor.bar@3.1;

import vendor.foovendor.foo@2.2::IFooBar;
import vendor.foovendor.foo@2.2::IFooFoo;

import IBar;
import IFoo;
Interface names
Interface names must start with an I, followed by an UpperCamelCase/PascalCase name. An interface with name IFoo must be defined in the file IFoo.hal. This file can contain definitions only for the IFoo interface (the interface INAME should be in INAME.hal).

Functions
For function names, arguments, and return variable names, use lowerCamelCase. Example:

open(INfcClientCallback clientCallback) generates (int32_t retVal);
oneway pingAlive(IFooCallback cb);
Struct and union field names
For struct or union field names, use lowerCamelCase. Example:

struct FooReply {

```
    vec<uint8_t> replyData;
}
```

Type names

Type names refer to struct or union definitions, enum type definitions, and typedefs. For these name, use UpperCamelCase/PascalCase. Examples:

```
enum NfcStatus : int32_t {
    /*...*/
};
struct NfcData {
    /*...*/
};
```

Enum values

Enum values should be UPPER_CASE_WITH_UNDERSCORES. When passing enum values as function arguments and returning them as function returns, use the actual enum type (not the underlying integer type). Example:

```
enum NfcStatus : int32_t {
    HAL_NFC_STATUS_OK               = 0,
    HAL_NFC_STATUS_FAILED           = 1,
    HAL_NFC_STATUS_ERR_TRANSPORT    = 2,
    HAL_NFC_STATUS_ERR_CMD_TIMEOUT  = 3,
    HAL_NFC_STATUS_REFUSED          = 4
};
```

Note: The underlying type of an enum type is explicitly declared after the colon. As it isn't compiler dependent, using the actual enum type is clearer.

For fully qualified names for enum values, a colon is used between the enum type name and the enum value name:

```
PACKAGE-NAME::UDT[.UDT[.UDT[…]]:ENUM_VALUE_NAME
```

There must not be spaces inside a fully qualified name. Use a fully qualified name only when necessary and omit unnecessary parts. Example:

```
android.hardware.foo@1.0::IFoo.IFooInternal.FooEnum:ENUM_OK
```

Comments

For a single line comment, //, /* */, and /** */ are fine.

```
// This is a single line comment
/* This is also single line comment */
/** This is documentation comment */
```

Use /* */ for comments. While HIDL supports // for comments, they are discouraged because they don't appear in generated output.
Use /** */ for generated documentation. These can be applied only to type, method, field, and enum value declarations. Example:

```
/** Replied status */
enum TeleportStatus {
    /** Object entirely teleported. */
    OK              = 0,
    /** Methods return this if teleportation is not completed. */
    ERROR_TELEPORT  = 1,
    /**
     * Teleportation could not be completed due to an object
     * obstructing the path.
     */
    ERROR_OBJECT    = 2,
    ...
}
```

Start multi-line comments with /** on a separate line. Use * at the start of each line. End the comment with */ on a separate line, aligning the asterisks. Example:

```
/**
 * My multi-line
 * comment
 */
```

Licensing notice and changelogs should start a new line with /* (a single asterisk), use * at the beginning of each line, and place */ on the last line all on its own (asterisks should align). Example:

```
/*
 * Copyright (C) 2017 The Android Open Source Project
 * ...
 */

/*
 * Changelog:
 * ...
 */
```

File comments

Start each file with the appropriate licensing notice. For core HALs, this should be the AOSP Apache license in development/docs/copyright-templates/c.txt. Remember to update the year and use /* */ style multi-line comments as explained above.

You can optionally place an empty line after the license notice, followed by a changelog/versioning information. Use /* */ style multi-line comments as explained above, place the empty line after the changelog, then follow with the package declaration.

TODO comments

TODOs should include the string TODO in all caps followed by a colon. Example:

```
// TODO: remove this code before foo is checked in.
```

TODO comments are allowed only during development; they must not exist in published interfaces.

Interface and function comments (docstrings)

Use /** */ for multi-line and single line docstrings. Don't use // for docstrings.

Docstrings for interfaces should describe general mechanisms of the interface, design rationale, purpose, etc. Docstrings for functions should be specific to the function (package-level documentation goes in a README file in the package directory).

```
/**
 * IFooController is the controller for foos.
 */
interface IFooController {
    /**
     * Opens the controller.
     *
     * @return status HAL_FOO_OK if successful.
     */
    open() generates (FooStatus status);

    /** Close the controller. */
    close();
};
```

You must add @params and @returns for each parameter/return value:

@param must be added for each parameter. It should be followed by the name of the parameter then the docstring.
@return must be added for each return value. It should be followed by the name of the return value then the docstring.
Example:

```
/**
 * Explain what foo does.
 *
 * @param arg1 explain what arg1 is
 * @param arg2 explain what arg2 is
 * @return ret1 explain what ret1 is
 * @return ret2 explain what ret2 is
 */
foo(T arg1, T arg2) generates (S ret1, S ret2);
```

Formatting rules

General formatting rules include:

Line length. Each line of text should be at most 100 columns long.
Whitespaces. No trailing whitespace on lines; empty lines must not contain whitespaces.
Spaces vs. tabs. Use only spaces.
Indent size. Use 4 spaces for blocks and 8 spaces for line wraps
Bracing. Except for annotation values, an open brace goes on the same line as preceding code but a close brace and the following semicolon occupies the entire line. Example:

```
interface INfc {
    close();
};
```

Package declaration
Package declaration should be at the top of the file after the license notice, should occupy the entire line, and shouldn't be indented. Packages are declared using the following format (for name formatting, see Package names):

```
package PACKAGE-NAME;
```

Example:

```
package android.hardware.nfc@1.0;
```

Function declarations
Function name, parameters, generates, and return values should be on the same line if they fit. Example:

```
interface IFoo {
    /** ... */
    easyMethod(int32_t data) generates (int32_t result);
};
```

If they don't fit on the same line, attempt to put parameters and return values in the same indent level and distinguish generate to help the reader quickly see the parameters and return values. Example:

```
interface IFoo {
    suchALongMethodThatCannotFitInOneLine(int32_t theFirstVeryLongParameter,
                                          int32_t anotherVeryLongParameter);
    anEvenLongerMethodThatCannotFitInOneLine(int32_t theFirstLongParameter,
                                             int32_t anotherVeryLongParameter)
                                  generates (int32_t theFirstReturnValue,
                                             int32_t anotherReturnValue);
    superSuperSuperSuperSuperSuperSuperLongMethodThatYouWillHateToType(
            int32_t theFirstVeryLongParameter, // 8 spaces
            int32_t anotherVeryLongParameter
        ) generates (
            int32_t theFirstReturnValue,
            int32_t anotherReturnValue
        );
    /* method name is even shorter than 'generates' */
    foobar(AReallyReallyLongType aReallyReallyLongParameter,
           AReallyReallyLongType anotherReallyReallyLongParameter)
        generates (ASuperLongType aSuperLongReturnValue, // 4 spaces
                   ASuperLongType anotherSuperLongReturnValue);
}
```

Additional details:

An open parenthesis is always on the same line as the function name.
No spaces between the function name and the open parenthesis.
No spaces between the parentheses and parameters except when there are line feeds between them.
If generates is on the same line as the previous closing parenthesis, use a preceding space. If generates is on the same line as the next open parenthesis, follow with a space.
Align all parameters and return values (if possible).
Default indentation is 4 spaces.
Wrapped parameters are aligned to the first parameters on the previous line, otherwise they have an 8-space indent.
Annotations
Use the following format for annotations:

@annotate(keyword = value, keyword = {value, value, value})
Sort annotations in alphabetical order, and use spaces around equal signs. Example:

```
@callflow(key = value)
@entry
@exit
```

Ensure an annotation occupies the entire line. Examples:

```
/* Good */
@entry
@exit

/* Bad */
@entry @exit
```

If annotations cannot fit on the same line, indent with 8 spaces. Example:

```
@annotate(
        keyword = value,
        keyword = {
                value,
                value
        },
        keyword = value)
```

If the entire value array cannot fit in the same line, put line breaks after open braces { and after each comma inside the array. Place closing parenthesis immediately after the last value. Don't put the braces if there is only one value.

If the entire value array can fit in the same line, don't use spaces after open braces and before closing braces and use one space after each comma. Examples:

```
/* Good */
@callflow(key = {"val", "val"})

/* Bad */
@callflow(key = { "val","val" })
```

There must NOT be empty lines between annotations and the function declaration. Examples:

```
/* Good */
@entry
foo();

/* Bad */
@entry

foo();
```

Enum declarations
Use the following rules for enum declarations:

If enum declarations are shared with another package, put the declarations in types.hal rather than embedding inside an interface.
Use a space before and after the colon, and space after the underlying type before the open brace.
The last enum value might not have an extra comma.
Struct declarations
Use the following rules for struct declarations:

If struct declarations are shared with another package, put the declarations in types.hal rather than embedding inside an interface.
Use a space after the struct type name before the open brace.
Align field names (optional). Example:

```
struct MyStruct {
    vec<uint8_t>   data;
    int32_t        someInt;
}
```

Array declarations
Don't put spaces between the following:

Element type and open square bracket.
Open square bracket and array size.
Array size and close square bracket.
Close square bracket and the next open square bracket, if more than one dimension exists.
Examples:

/* Good */
int32_t[5] array;

/* Good */
int32_t[5][6] multiDimArray;

/* Bad */
int32_t [5] [6] array;
Vectors
Don't put spaces between the following:

vec and open angle bracket.
Open angle bracket and element type (Exception: element type is also a vec).
Element type and close angle bracket (Exception: element type is also a vec).
Examples:

/* Good */
vec<int32_t> array;

/* Good */
vec<vec<int32_t>> array;

/* Good */
vec< vec<int32_t> > array;

/* Bad */
vec < int32_t > array;

/* Bad */
vec < vec < int32_t > > array;

Here's a **simple explanation** of all the rules and examples you shared about writing clean and correct **HIDL code**:

🧠 What Is This About?

This is a **style guide** for writing **`.hal`** files in HIDL — just like how we follow coding standards in C++ or Java. Google defines these rules so that **everyone writes HAL code in the same way**, making it easier to read, review, and maintain.

🗂️ 1. Directory and File Naming Rules

✅ Folder Structure

```bash
hardware/interfaces/
  foo/
    1.0/
      Android.mk
      IFoo.hal
      IFooClientCallback.hal
      types.hal
```

- Use lowercase for folders (like **`foo`**).

- Interfaces (`IFoo`) use `PascalCase` and start with `I`.
- `types.hal` stores shared data types (like `structs`, `enums`).

📦 2. Package Naming

```hidl
package android.hardware.foo@1.0;
```

- `android.hardware` → system HAL
- `vendor.qualcomm.hardware.foo@1.0` → vendor HAL
- Package name = folder path

✅ **No spaces**, use **lowercase**, use `@VERSION`.

📥 3. Imports

```hidl
import android.hardware.bar@1.0::IBar;
import vendor.xyz.foo@1.2::types;

import IBaz;  // same package
```

- Use **fully qualified names** for other packages
- Use short names for same-package imports
- Group and sort them:
  1. `android.hardware`
  2. `vendor.VENDOR`
  3. local imports

✍️ 4. Function Style

✅ Function name and parameters

```hidl
foo(int32_t a, int32_t b) generates (int32_t result);
```

❗ If too long, break lines:

```hidl
veryLongMethod(int32_t reallyLongParamA,
               int32_t reallyLongParamB)
        generates (int32_t outputA,
                   int32_t outputB);
```

- `4 spaces` for blocks
- `8 spaces` for wrapped lines

📘 5. Comments & Docstrings

```hidl
/**
 * Describe what this interface does.
 */
interface IFoo {
    /**
```

```
     * Opens the device.
     *
     * @param x What this input means
     * @return status Success/failure code
     */
    open(int32_t x) generates (int32_t status);
};
```

- Use `/** ... */` for **docstrings**
- Use `/* ... */` for **normal comments**
- Use `//` only during development (`// TODO:`)

📇 6. Naming Conventions

Thing	Format	Example
Interface name	PascalCase + `I`	`IFoo`, `INfcCallback`
Struct field	lowerCamelCase	`someData`
Enum name	PascalCase	`NfcStatus`
Enum value	UPPER_CASE	`STATUS_OK`
Type name	PascalCase	`FooData`
Function name	lowerCamelCase	`openDevice()`
Filenames	Match interface	`IFoo.hal`, `types.hal`

📐 7. Formatting Rules

Rule	Example
Line length ≤ 100 chars	Keep lines short and readable
Use 4 spaces (no tabs)	Indent each block
Braces on same line	`interface IFoo { ... };`
Empty line after package	
Align parameters	Line up long param/return lists

📚 8. Example Summary

📄 `types.hal`

```hidl
/*
 * Copyright (C) ...
 */

package android.hardware.foo@1.0;

/** Status of operation */
enum FooStatus : int32_t {
    OK,
    ERR_UNKNOWN = -1,
};

struct FooData {
    vec<uint8_t>  payload;
    int32_t[10]   meta;
};
```

📄 `IFoo.hal`

```hidl
/*
 * Copyright (C) ...
 */

package android.hardware.foo@1.0;
```

```
import android.hardware.bar@1.0::IBar;
import IBaz;
import IFooClientCallback;

/**
 * IFoo controls Foo device.
 */
interface IFoo {
    /**
     * Simple operation.
     *
     * @return result 0 on success.
     */
    foo() generates (FooStatus result);

    /**
     * Restart controller.
     *
     * @param bar a bar interface
     * @return result FooStatus code
     */
    powerCycle(IBar bar) generates (FooStatus result);

    /** Just trigger baz. */
    baz();

    /**
     * The bar operation.
     *
     * @param clientCallback callback interface
     * @param baz Baz object
     * @param data Foo data
     */
    bar(IFooClientCallback clientCallback,
        IBaz baz,
        FooData data);
};
```

✅ Best Practices to Remember

- Keep shared types in **`types.hal`**
- Use **`@param`** and **`@return`** in docstrings
- Use consistent indentation
- Always start with license header
- No TODOs in production **`.hal`** files

Would you like a real interface + types.hal example written from scratch following all these rules?

37. HIDL C++

bookmark_border
Android 8 re-architects the Android OS to define clear interfaces between the device-independent Android platform and device- and vendor-specific code. Android already defines many such interfaces in the form of HAL interfaces, defined as C headers in hardware/libhardware. HIDL replaces these HAL interfaces with stable, versioned interfaces, which can be client- and server-side HIDL interfaces in C++ (described below) or Java.

The pages in this section describe C++ implementations of HIDL interfaces, including details about the files auto-generated from the HIDL .hal files by the hidl-gen compiler, how these files are packaged, and how to integrate these files with the C++ code that uses them.

Client and server implementations
HIDL interfaces have client and server implementations:

A client of a HIDL interface is the code that uses the interface by calling methods on it.
A server is an implementation of a HIDL interface that receives calls from clients and returns results (if necessary).
In transitioning from libhardware HALs to HIDL HALs, the HAL implementation becomes the server and the process calling into the HAL becomes the client. Default implementations can serve both passthrough and binderized HALs, and can change over time:

Figure 1. Development progression for legacy HALs.

Create the HAL client
Start by including the HAL libraries in the makefile:

Make: LOCAL_SHARED_LIBRARIES += android.hardware.nfc@1.0
Soong: shared_libs: [..., android.hardware.nfc@1.0]
Next, include the HAL header files:

```
#include <android/hardware/nfc/1.0/IFoo.h>
…
// in code:
sp<IFoo> client = IFoo::getService();
client->doThing();
```

Create the HAL server
To create the HAL implementation, you must have the .hal files that represent your HAL and have already generated makefiles for your HAL using -Lmakefile or -Landroidbp on hidl-gen (./hardware/interfaces/update-makefiles.sh does this for internal HAL files and is a good reference). When transferring over HALs from libhardware, you can do a lot of this work easily using c2hal.

To create the necessary files to implement your HAL:

```
PACKAGE=android.hardware.nfc@1.0
LOC=hardware/interfaces/nfc/1.0/default/
m -j hidl-gen
hidl-gen -o $LOC -Lc++-impl -randroid.hardware:hardware/interfaces \
    -randroid.hidl:system/libhidl/transport $PACKAGE
hidl-gen -o $LOC -Landroidbp-impl -randroid.hardware:hardware/interfaces \
    -randroid.hidl:system/libhidl/transport $PACKAGE
```

For the HAL to work in passthrough mode, you must have the function HIDL_FETCH_IModuleName residing in /(system|vendor|...)/lib(64)?/hw/android.hardware.package@3.0-impl(OPTIONAL_IDENTIFIER).so where OPTIONAL_IDENTIFIER is a string identifying the passthrough implementation. The passthrough mode requirements are met automatically by the above commands, which also create the android.hardware.nfc@1.0-impl target, but any extension can be used. For instance android.hardware.nfc@1.0-impl-foo uses -foo to differentiate itself.

If a HAL is a minor version or an extension of another HAL, the base HAL should be used to name this binary. For instance, android.hardware.graphics.mapper@2.1 implementations should still be in a binary called android.hardware.graphics.mapper@2.0-impl(OPTIONAL_IDENTIFIER). Usually, the OPTIONAL_IDENTIFIER here would include the actual HAL version. By naming the binary like this, 2.0 clients can retrieve it directly, and 2.1 clients can upcast the implementation.

Next, fill out the stubs with functionality and setup a daemon. Example daemon code (supporting passthrough):


```
#include <hidl/LegacySupport.h>

int main(int /* argc */, char* /* argv */ []) {
    return defaultPassthroughServiceImplementation<INfc>("nfc");
}
defaultPassthroughServiceImplementation calls dlopen() for the provided -impl library and provides it as a binderized service. Example daemon code (for pure binderized service):


int main(int /* argc */, char* /* argv */ []) {
    // This function must be called before you join to ensure the proper
    // number of threads are created. The threadpool never exceeds
    // size one because of this call.
    ::android::hardware::configureRpcThreadpool(1 /*threads*/, true /*willJoin*/);

    sp<INfc> nfc = new Nfc();
    const status_t status = nfc->registerAsService();
    if (status != ::android::OK) {
        return 1; // or handle error
    }

    // Adds this thread to the threadpool, resulting in one total
    // thread in the threadpool. We could also do other things, but
    // would have to specify 'false' to willJoin in configureRpcThreadpool.
    ::android::hardware::joinRpcThreadpool();
    return 1; // joinRpcThreadpool should never return
}
This daemon usually lives in $PACKAGE + "-service-suffix" (for example, android.hardware.nfc@1.0-service), but it could be anywhere. The sepolicy for a specific class of HALs is the attribute hal_<module> (for instance, hal_nfc). This attribute must be applied to the daemon that runs a particular HAL (if the same process serves multiple HALs, multiple attributes can be applied to it).
```

Here's a **simple explanation** of HIDL C++ implementation and how **HAL clients and servers** work in Android 8+:

🧠 What is HIDL?

HIDL stands for **HAL Interface Definition Language**.

- Before Android 8, HALs were just **C header files** (`hardware/libhardware`) with **no strict versioning**.
- From Android 8 onward, HIDL was introduced to:
  - **Version** the HALs
  - Define **clear interfaces**
  - Support **stable communication** between Android Framework and device-specific code

---

🧩 Key Components

✅ `.hal` File

This is the HIDL file you write (like `INfc.hal`) that defines functions, structs, enums, etc.

👥 Roles: Client and Server

- **Client:** Calls the functions (e.g., Android Framework or apps)
- **Server:** Implements the functions (e.g., vendor HAL code)

Example

```cpp
// Client-side
sp<INfc> nfc = INfc::getService();  // Binds to the service
nfc->open();                        // Calls the HAL function

// Server-side
class Nfc : public INfc {
  Return<void> open() override {
    // Your implementation here
  }
};
```

🛠️ Client Setup

To use a HAL in your code (as a client):

1. Add the HAL to your **makefile**:
   - **`LOCAL_SHARED_LIBRARIES += android.hardware.nfc@1.0`**
   - or in **`Android.bp`**:

     `shared_libs: [ "android.hardware.nfc@1.0", ... ]`
2. Include header:

   ```cpp
   #include <android/hardware/nfc/1.0/INfc.h>
   using android::hardware::nfc::V1_0::INfc;
   ```

3. Use the service:

   ```cpp
   sp<INfc> client = INfc::getService();  // Binder call
   client->doThing();
   ```

🔧 Server Setup (HAL Implementation)

1. **Generate the code** using **`hidl-gen`**:

```sh
PACKAGE=android.hardware.nfc@1.0
LOC=hardware/interfaces/nfc/1.0/default/

hidl-gen -o $LOC -Lc++-impl -randroid.hardware:hardware/interfaces \
         -randroid.hidl:system/libhidl/transport $PACKAGE

hidl-gen -o $LOC -Landroidbp-impl -randroid.hardware:hardware/interfaces \
         -randroid.hidl:system/libhidl/transport $PACKAGE
```

This generates:

- **`Nfc.h`** (C++ stub to fill)
- **`Android.bp`** for default implementation
- **`android.hardware.nfc@1.0-impl.so`** library (passthrough)

🚀 Run as a Daemon (Server Process)

Option 1: Passthrough Daemon

```cpp
#include <hidl/LegacySupport.h>

int main() {
    return defaultPassthroughServiceImplementation<INfc>("nfc");
}
```

- Uses **`dlopen()`** to load **`android.hardware.nfc@1.0-impl.so`**
- Acts like old HALs (**`libhardware`**) but cleaner

Option 2: Binderized Daemon

```cpp
int main() {
    ::android::hardware::configureRpcThreadpool(1, true);

    sp<INfc> nfc = new Nfc();  // Your HAL implementation class
    if (nfc->registerAsService() != android::OK) {
        return 1;
    }

    ::android::hardware::joinRpcThreadpool();  // Serve forever
    return 1;
}
```

- Uses binder (IPC)
- Cleaner separation and sandboxing
- Lives in **`/vendor/bin/hw/android.hardware.nfc@1.0-service`**

📦 Naming and Library Notes

- Generated **`.so`** library:
  `android.hardware.nfc@1.0-impl.so`
- Service binary:
  `android.hardware.nfc@1.0-service`
- You can suffix versions like **`@1.0-impl-qcom`** for custom variants

🔐 SEPolicy Label

For **`nfc`**, your service needs:

```c
type hal_nfc, domain;
```

Apply this label to your service daemon (**`.te`** + **`.rc`** files).

🧾 TL;DR Flow

```text
1. Write .hal file
2. Run hidl-gen → generate stubs and build files
3. Fill stubs (server)
4. Add client calls (framework/app)
5. Register service with binder or passthrough
6. Use SEPolicy to secure it
7. Service runs and responds to client
```

Would you like a real-world example of a **`INfc.hal`**, its generated C++ server class, and a simple client that calls it?

38. ↳
PACKAGE=android.hardware.nfc@1.0 LOC=hardware/interfaces/nfc/1.0/default/ hidl-gen -o $LOC -Lc++-impl -randroid.hardware:hardware/interfaces \\ -randroid.hidl:system/libhidl/transport $PACKAGE hidl-gen -o $LOC -Landroidbp-impl -randroid.hardware:hardware/interfaces \\ -randroid.hidl:system/libhidl/transport $PACKAGE
explain this

Sure! Let's break down this HIDL-related command in **very simple terms**.

🧩 What does this command do?

This command runs the **`hidl-gen` tool** to **generate C++ code and Android build files** for a HIDL interface (`.hal` file).

It's like saying:

> "Hey, I've written the HIDL interface file (`.hal`) — now please create the necessary C++ code and Android.bp file so I can use or implement it in my project."

🧱 Full command breakdown

```bash
PACKAGE=android.hardware.nfc@1.0
LOC=hardware/interfaces/nfc/1.0/default/

# Generate C++ stub files for implementation
hidl-gen -o $LOC -Lc++-impl \
  -randroid.hardware:hardware/interfaces \
  -randroid.hidl:system/libhidl/transport \
  $PACKAGE

# Generate Android.bp file for the implementation
hidl-gen -o $LOC -Landroidbp-impl \
  -randroid.hardware:hardware/interfaces \
  -randroid.hidl:system/libhidl/transport \
  $PACKAGE
```

🔍 What each part means

✅ `PACKAGE=android.hardware.nfc@1.0`

This defines the name of the interface you're generating code for — in this case, the NFC HAL interface.

✅ `LOC=hardware/interfaces/nfc/1.0/default/`

This is the output location where generated files will be placed.

🔧 `hidl-gen` command details

🛠️ 1. Generate C++ implementation stubs:

```bash
hidl-gen -o $LOC -Lc++-impl ...
```

- **`-Lc++-impl`** means: generate **C++ class files** that you will **fill in with logic**.
- Output is stored in **`$LOC`**, like:

  ```swift
  hardware/interfaces/nfc/1.0/default/Nfc.cpp
  hardware/interfaces/nfc/1.0/default/Nfc.h
  ```

🛠️ **2. Generate Android.bp file:**

```bash
hidl-gen -o $LOC -Landroidbp-impl ...
```

- **`-Landroidbp-impl`** means: generate the **Android.bp** build file to compile the HAL implementation (Nfc.cpp).
- It produces a **`Android.bp`** file like:

  ```bp
  cc_library_shared {
      name: "android.hardware.nfc@1.0-impl",
      ...
  }
  ```

🧩 What does `-r` mean?

```bash
-randroid.hardware:hardware/interfaces \
-randroid.hidl:system/libhidl/transport
```

This tells **`hidl-gen`**:

> "Hey, when you see the package name **`android.hardware`** or **`android.hidl`**, they live at these locations in my source tree."

So:

- **`android.hardware`** → **`hardware/interfaces`**
- **`android.hidl`** → **`system/libhidl/transport`**

This is needed so it can **resolve imported types**, like:

```hal
import android.hidl.base@1.0::IBase;
```

📦 After this, you get:

- C++ files like **`Nfc.h`**, **`Nfc.cpp`** – empty functions you need to implement.
- Android.bp file to compile your code as **`android.hardware.nfc@1.0-impl`**.

✅ Summary

Part	What it does
`-Lc++-impl`	Generates C++ stub classes (**`Nfc.cpp`**, **`Nfc.h`**)
`-Landroidbp-impl`	Generates **`Android.bp`** to build the HAL
`-r`	Resolves where packages like **`android.hardware`** or **`android.hidl`** are in the AOSP source tree
`$PACKAGE`	The name of the HIDL interface you're working with

Would you like a real example of:

- **`.hal`** file input
- and what gets generated (like the **`Nfc.cpp`**)?

39. explain with example

Let's go step-by-step with a **real working example** of generating a HIDL HAL interface and its C++ implementation using **`hidl-gen`**.

✅ Step-by-Step HIDL HAL Interface Example

📁 We'll create a HAL: `android.hardware.led@1.0::ILed`

📝 1. `.hal` file (interface definition)

File: **`hardware/interfaces/led/1.0/ILed.hal`**

```hal
package android.hardware.led@1.0;

/**
 * Simple LED interface to turn on/off an LED.
 */
interface ILed {
    /**
     * Turn the LED on.
     * @return success true if the LED was turned on.
     */
    turnOn() generates (bool success);

    /**
     * Turn the LED off.
     * @return success true if the LED was turned off.
     */
    turnOff() generates (bool success);
};
```

🔧 Step 2: Generate C++ Stub Code using `hidl-gen`

💻 Command to run

```bash
PACKAGE=android.hardware.led@1.0
LOC=hardware/interfaces/led/1.0/default/

hidl-gen -o $LOC -Lc++-impl \
  -randroid.hardware:hardware/interfaces \
  -randroid.hidl:system/libhidl/transport \
  $PACKAGE

hidl-gen -o $LOC -Landroidbp-impl \
  -randroid.hardware:hardware/interfaces \
  -randroid.hidl:system/libhidl/transport \
  $PACKAGE
```

📂 Step 3: What Gets Generated?

➕ Files created in `hardware/interfaces/led/1.0/default/`

`Led.h` (Class header)

```cpp
#ifndef ANDROID_HARDWARE_LED_V1_0_LED_H
#define ANDROID_HARDWARE_LED_V1_0_LED_H

#include <android/hardware/led/1.0/ILed.h>
#include <hidl/MQDescriptor.h>
#include <hidl/Status.h>

namespace android {
namespace hardware {
namespace led {
```

```
namespace V1_0 {
namespace implementation {

using ::android::hardware::led::V1_0::ILed;
using ::android::hardware::hidl_array;
using ::android::hardware::hidl_memory;
using ::android::hardware::Return;
using ::android::hardware::Void;
using ::android::sp;

struct Led : public ILed {
    // Methods from ::android::hardware::led::V1_0::ILed follow.
    Return<bool> turnOn() override;
    Return<bool> turnOff() override;
};

}  // namespace implementation
}  // namespace V1_0
}  // namespace led
}  // namespace hardware
}  // namespace android

#endif  // ANDROID_HARDWARE_LED_V1_0_LED_H
```

`Led.cpp` (Stub implementation)

```cpp
#include "Led.h"

namespace android {
namespace hardware {
namespace led {
namespace V1_0 {
namespace implementation {

Return<bool> Led::turnOn() {
    // TODO: Add actual hardware control logic
    return true;
}

Return<bool> Led::turnOff() {
    // TODO: Add actual hardware control logic
    return true;
}

}  // namespace implementation
}  // namespace V1_0
}  // namespace led
}  // namespace hardware
}  // namespace android
```

🏗️ Step 4: Generated Android.bp file

Generated in the same `default/` directory:

```bp
cc_library_shared {
    name: "android.hardware.led@1.0-impl",
    defaults: ["hidl_defaults"],
    proprietary: true,
    relative_install_path: "hw",
    srcs: [
        "Led.cpp",
    ],
    shared_libs: [
        "liblog",
        "libhidlbase",
        "libhidltransport",
        "libutils",
        "android.hardware.led@1.0",
    ],
}
```

🧠 Final Step: Add Service Daemon

File: **`android.hardware.led@1.0-service.cpp`**

```cpp
#include <android/hardware/led/1.0/ILed.h>
#include <hidl/LegacySupport.h>
#include "Led.h"

using android::hardware::led::V1_0::ILed;
using android::hardware::led::V1_0::implementation::Led;
using android::hardware::defaultPassthroughServiceImplementation;

int main() {
    return defaultPassthroughServiceImplementation<ILed>();
}
```

✅ Recap

Component	Purpose
`ILed.hal`	Defines the HIDL interface
`hidl-gen`	Generates stub C++ code + Android.bp
`Led.cpp` / **`Led.h`**	Stub implementation that you fill
`Android.bp`	Build configuration
`-service.cpp`	Main entry that registers the HAL with hwservicemanager

Would you like a full **folder structure and build guide** as seen in AOSP for this HAL (along with SELinux policy + manifest)?

40. ↳
hidl-gen -o $LOC -Lc++-impl \\ -randroid.hardware:hardware/interfaces \\ -randroid.hidl:system/libhidl/transport \\ $PACKAGE hidl-gen -o $LOC -Landroidbp-impl \\ -randroid.hardware:hardware/interfaces \\ -randroid.hidl:system/libhidl/transport \\ $PACKAGE
confused why we need -r explain with example

Great question. The `-r` flag in `hidl-gen` is **crucial** for resolving the locations of imported packages.

Let's break it down simply with **what it does**, **why it's needed**, and a **clear example**.

✅ What does `-r` mean in `hidl-gen`?

The `-r` flag tells `hidl-gen` **how to resolve the root directory of a package prefix**.

Syntax:

```ruby
-r <PACKAGE_PREFIX>:<DIRECTORY_PATH>
```

Meaning:

> Whenever the package name starts with `<PACKAGE_PREFIX>`, replace it with the directory `<DIRECTORY_PATH>` to find the actual `.hal` files.

🔍 Why do we need it?

Because `.hal` files often `import` other interfaces. For example:

```hal
import android.hardware.bar@1.0::IBar;
```

Now `hidl-gen` needs to know:

- Where is `android.hardware.bar@1.0` located?
- Which folder should it go to in the file system to find that `.hal` file?

That's where `-r` comes in.

📘 Example: You are generating `ILed.hal`

Suppose your file is:

hardware/interfaces/led/1.0/ILed.hal

```hal
package android.hardware.led@1.0;

import android.hardware.bar@1.0::IBar;

interface ILed {
    turnOn(IBar bar) generates (bool success);
};
```

Now you run:

```bash
hidl-gen -o hardware/interfaces/led/1.0/default/ -Lc++-impl \
  -randroid.hardware:hardware/interfaces \
  -randroid.hidl:system/libhidl/transport \
  android.hardware.led@1.0
```

Here's what happens:

Flag	Meaning
`-randroid.hardware:hardware/interfaces`	Any **`android.hardware.*`** package is located under **`hardware/interfaces`**
`-randroid.hidl:system/libhidl/transport`	Any **`android.hidl.*`** package is located under **`system/libhidl/transport`**

So when **`ILed.hal`** imports:

```hal
import android.hardware.bar@1.0::IBar;
```

`hidl-gen` looks for:

```swift
hardware/interfaces/bar/1.0/IBar.hal
```

because of:

```bash
-randroid.hardware:hardware/interfaces
```

If you **don't** provide **`-r`**, **`hidl-gen`** doesn't know where to find **`IBar.hal`**, and it will fail with:

```kotlin
ERROR: Could not find imported file for package android.hardware.bar@1.0
```

🧠 Key Rule:

> Each **`-r`** is a **mapping** from a **package prefix** to its **directory on disk**.

🔄 You can even add your own vendor HAL:

Let's say you write:

```hal
import vendor.sanjay.led@1.0::ILedHelper;
```

Then you **must** add:

```bash
-rvendor.sanjay:vendor/sanjay/interfaces
```

So **`hidl-gen`** can look into:

```swift
vendor/sanjay/interfaces/led/1.0/ILedHelper.hal
```

✅ Summary

Use Case	Need `-r`?
You **`import`** another package?	✅ Yes
Your **`.hal`** depends on core Android or vendor interfaces?	✅ Yes
All your types are in one **`.hal`** file?	❌ Might not need it, but safer to always include

---

Would you like me to show a **failing case without `-r`** and how it succeeds **with `-r`**?

41. yes

Perfect! Let's walk through a concrete **example** that shows what happens:

🔴 Case 1: Without `-r` → Failure

You have this **`ILed.hal`**:

📄 **File: `hardware/interfaces/led/1.0/ILed.hal`**

```hal
package android.hardware.led@1.0;

import android.hardware.light@2.0::ILight;

interface ILed {
    lightUp(ILight light) generates (bool success);
};
```

Now try to generate the implementation:

```bash
hidl-gen -o hardware/interfaces/led/1.0/default/ -Lc++-impl android.hardware.led@1.0
```

❌ Error:

```kotlin
ERROR: Could not find imported file for package android.hardware.light@2.0
```

This happened because:

- **`ILed.hal`** imports **`android.hardware.light@2.0::ILight`**
- **`hidl-gen`** doesn't know where to look for **`android.hardware.light@2.0`**

✅ Case 2: With `-r` → Success

Let's try again, but **correctly tell `hidl-gen`** where to find the imported package:

```bash
hidl-gen -o hardware/interfaces/led/1.0/default/ -Lc++-impl \
  -randroid.hardware:hardware/interfaces \
  android.hardware.led@1.0
```

✅ Now works! 🎉

Because:

- The **`-randroid.hardware:hardware/interfaces`** means:

  > If the package starts with **`android.hardware`**, go to **`hardware/interfaces`**

- So:
  - **`android.hardware.light@2.0::ILight`** →
  - → **`hardware/interfaces/light/2.0/ILight.hal`**

This allows the **`import android.hardware.light@2.0::ILight;`** to **resolve successfully**, and the stub classes for **`ILed`** will be generated.

✅ Bonus: Add multiple `-r` for other namespaces

If your `.hal` imports from:

- `android.hardware.*` → add `-randroid.hardware:hardware/interfaces`
- `android.hidl.*` → add `-randroid.hidl:system/libhidl/transport`
- `vendor.yourname.*` → add `-rvendor.yourname:vendor/yourname/interfaces`

🧠 Summary Table

Command	Result
`hidl-gen -o ... -Lc++-impl android.hardware.led@1.0`	❌ fails (no `-r` for imported package)
`hidl-gen -o ... -Lc++-impl -randroid.hardware:hardware/interfaces android.hardware.led@1.0`	✅ success

Let me know if you want a full working demo structure or a sample project for testing this live.

42. ↳
hidl-gen -o hardware/interfaces/led/1.0/default/ -Lc++-impl \\ -randroid.hardware:hardware/interfaces \\ android.hardware.led@1.0
in which case we should use hidl-gen and in which case we should module hidl_interface?

Great question! Let's clearly explain when to use:

✅ `hidl-gen` command

This is the **manual tool** used to generate code (interface stubs, default impl, etc.) from `.hal` files.

🔸 When should you use it?

You should use `hidl-gen` when:

- You're working **outside of AOSP build system** (e.g., testing things locally).
- You want to **manually generate**:
  - C++ interface headers (`-Lc++-headers`)
  - Implementation stubs (`-Lc++-impl`)
  - Android.bp files (`-Landroidbp`)
  - Makefiles (`-Lmakefile`)

🛠 Example:

```bash
hidl-gen -o vendor/myvendor/interfaces/led/1.0/default/ \
  -Lc++-impl \
  -rvendor.myvendor:vendor/myvendor/interfaces \
  vendor.myvendor.hardware.led@1.0
```

✅ `hidl_interface` Soong module

This is used **inside AOSP build system** (`Android.bp`) to:

- Automatically trigger `hidl-gen` behind the scenes.
- Build all artifacts for a `.hal` package.
- Create required modules like:
  - `android.hardware.foo@1.0`
  - `android.hardware.foo@1.0-impl`
  - `android.hardware.foo@1.0-service`
  - Stub header libraries
  - Symlink libraries, etc.

🔸 When should you use it?

Use `hidl_interface` in your `Android.bp` when:

- You're working **inside the AOSP build system**.
- You want your `.hal` to integrate cleanly and build automatically.

🛠 Example `Android.bp`:

```bp
hidl_interface {
    name: "android.hardware.led@1.0",
    root: "hardware/interfaces",
    srcs: [
        "ILed.hal",
    ],
    interfaces: [],
    gen_java: false,
}
```

Then just run:

```bash
m android.hardware.led@1.0
```

No need to run **`hidl-gen`** manually!

🧠 Summary Table

Use case	Use `hidl-gen`	Use `hidl_interface`
Testing HALs outside AOSP	✅ Yes	❌ No
Manual stub/header generation	✅ Yes	❌ No
Integrating into full Android build	❌ No	✅ Yes
Vendor/Board HAL for product	❌ No	✅ Yes
Automatically generate **`-impl`**, **`-service`**, **`-bp`** files	❌ No	✅ Yes
Only need headers for experimentation	✅ Yes	❌ No

🔁 Typical Workflow

1. **Draft `.hal` interface**
2. Use **`hidl-gen`** to **test and see generated code**
3. Once finalized, **declare with `hidl_interface` in Android.bp**
4. Let Soong manage build and packaging

Let me know if you want a sample working HAL project that uses both styles!

43. ⁘
explain in simple word

This image shows how **Android HALs (Hardware Abstraction Layers)** evolved from **legacy HALs** to the **HIDL model** across Android versions, especially after **Project Treble** (Android 8+). Let me explain each part **in simple terms**, using this image.

🔷 Top Layer: Framework

This is the **Android system** (Java/Kotlin or C++ framework) which wants to talk to hardware.
Example: the Camera app needs camera hardware → it calls into the Framework.

🔶 Orange Area: Legacy HAL

Before Android 8:

- The **Framework directly linked to the HAL implementation** using `.so` files (shared libraries).
- There was **no versioning**.
- It was all tightly coupled.
- All HALs and Android Framework lived in the **same process or binary** (tight integration).

Problem? You can't upgrade the system or HAL independently. Very hard for vendors to update just drivers or HALs.

🟡 Middle Area: HIDL HALs (Android 8+)

After **Project Treble**, Android introduced HIDL to create a **clear separation** between:

- Android system (framework)
- HAL implementations (vendor code)

HIDL supports two modes:

✅ HIDL Passthrough

- Like Legacy HAL, but **wrapped in HIDL interface**.
- Framework uses the HAL **in the same process**, via HIDL C++.
- Uses `dlopen()` to load the HAL `.so` (shared library) at runtime.
- Good for minimal migration from legacy HALs.

📌 **Used when**: the HAL is **simple**, doesn't need full binder IPC.

✅ HIDL Binderized

- The HAL is now a **separate service** running in **its own process**.
- Framework uses **Binder IPC** to talk to the HAL.
- **More robust**: no crashing the system if HAL fails.
- HAL runs like a **system service** (e.g. `android.hardware.camera@1.0-service`)

📌 **Used when**: the HAL is **complex**, performance isn't impacted, or process separation is needed (recommended by Google).

🟩 Green Boxes: Default Implementations

- HIDL supports providing a **default implementation** (reference HAL).

- Can be:
  - Used for emulation/testing.
  - Fallback if vendor doesn't provide their own.
- These sit between framework and the HAL implementation.

🔴 Bottom Row: Vendor Implementations

These are the **real HALs written by device makers**:

- Legacy HAL: pre-Treble (tight coupling)
- Vendor HAL: post-Treble (separate service/process)

💡 Overall Evolution Summary

Stage	Description
Legacy HAL	Framework directly calls vendor HAL. No versioning. All-in-one.
HIDL Passthrough	Wrapped in HIDL, still loaded in same process (via **`dlopen`**)
HIDL Binderized	Full separation via Binder IPC, runs in own process. Clean boundary.

Let me know if you want me to **map this to a real example** like NFC or Camera HAL.

44. in picture how may column you can see?

In the picture, there are **four vertical columns**, each showing a **different stage or architecture** of Android HAL evolution. Here's a quick breakdown:

✅ Column 1: Legacy HAL

- No HIDL.
- Framework directly uses the legacy HAL.
- Implementation lives in the same process.

✅ Column 2: HIDL (Passthrough)

- Framework uses HIDL interface.
- HAL is still loaded via `dlopen()` in the same process.
- Minimal separation.

✅ Column 3: HIDL (Binderized)

- Framework uses HIDL interface.
- HAL is **a separate binderized service**.
- Strong isolation and clean IPC boundary.

✅ Column 4: HIDL (Binderized with HW Service Vendor Implementation)

- Similar to column 3.
- But here, the HAL implementation is done entirely by **vendor** as a separate service process.

So in total, there are **4 columns**, showing the evolution:

1. Legacy HAL
2. HIDL Passthrough
3. HIDL Binderized (with default implementation)
4. HIDL Binderized (fully vendor-implemented)

Let me know if you'd like a table comparing these 4!

45. what do you mean by default implementation here in pic

Great question! Let's explain **"default implementation"** in the context of the diagram.

🔹 What is "default implementation" in the picture?

In the **middle two columns** of the diagram ("HIDL passthrough" and "HIDL binderized"), you see a green box labeled **Default Implementation**.

This means:

> Google (AOSP) provides a **default reference implementation** of the HIDL interface — usually simple, maybe even stubbed — that vendors can **reuse**, extend, or override.

🔧 Why is it called "default"?

Because it's the **default version** of the HAL provided by the AOSP or the platform team. It usually lives in:

```pgsql
hardware/interfaces/<hal-name>/<version>/default/
```

It's called **default** because:

- It's the **default fallback** if a vendor doesn't provide their own.
- Vendors can **build on top of it** or **replace** it with their own implementation (as in the 4th column of the diagram).

🛠️ Example:

Suppose you're dealing with `android.hardware.nfc@1.0`.

- AOSP might provide a `Nfc.cpp` file in:

  ```swift
  hardware/interfaces/nfc/1.0/default/
  ```

- This is the **default implementation** of `INfc`.

If you're a vendor and you don't have any special needs, you could use this directly. Otherwise, you can:

- **Copy it** to your own source directory
- **Extend** or **modify** the behavior
- And ship it in your own HAL `.so` or `.service`

✅ In the picture:

Column	Default Implementation Role
HIDL Passthrough	Directly loaded via `dlopen()` in the same process
HIDL Binderized	Registered as a binderized service using `registerAsService()`
Vendor Impl (last column)	Vendor replaces AOSP's default with their own

Let me know if you'd like a real path/code example to see this in AOSP!

46. Packages

bookmark_border
Note: This section uses sample .hal files to illustrate how HIDL language constructs map to C++.
With few exceptions, HIDL interface packages are located in hardware/interfaces or the vendor/ directory. The hardware/interfaces top-level maps directly to the android.hardware package namespace; the version is a subdirectory under the package (not interface) namespace.

The hidl-gen compiler compiles the .hal files into a set of a .h and .cpp files. From these autogenerated files a shared library that client/server implementations link against is built. The Android.bp file that builds this shared library is autogenerated by the hardware/interfaces/update-makefiles.sh script. Every time you add a new package to hardware/interfaces, or add/remove .hal files to/from an existing package, you must rerun the script to ensure the generated shared library is up-to-date.

For example, the IFoo.hal sample file should be located in hardware/interfaces/samples/1.0. The sample IFoo.hal file creates an IFoo interface in the samples package:

```
package android.hardware.samples@1.0;
interface IFoo {
   struct Foo {
     int64_t someValue;
     handle  myHandle;
   };

   someMethod() generates (vec<uint32_t>);
   anotherMethod(Foo foo) generates (int32_t ret);
};
```

Generated files
Autogenerated files in a HIDL package are linked into a single shared library with the same name as the package (for example, android.hardware.samples@1.0). The shared library also exports a single header, IFoo.h, which can be included by clients and servers. Using the hidl-gen compiler with the IFoo.hal interface file as an input, binderized mode has the following autogenerated files:

Files
generated by compiler

Figure 1. Files generated by compiler.

IFoo.h. Describes the pure IFoo interface in a C++ class; it contains the methods and types defined in the IFoo interface in the IFoo.hal file, translated to C++ types where necessary. Doesn't contain details related to the RPC mechanism (for example, HwBinder) used to implement this interface. The class is namespaced with the package and version, for example, ::android::hardware::samples::IFoo::V1_0. Both clients and servers include this header: Clients for calling methods on it and servers for implementing those methods.
IHwFoo.h. Header file that contains declarations for functions that serialize data types used in the interface. Developers should never include his header directly (it doesn't contain any classes).
BpHwFoo.h. A class that inherits from IFoo and describes the HwBinder proxy (client-side) implementation of the interface. Developers should never refer to this class directly.
BnHwFoo.h. A class that holds a reference to an IFoo implementation and describes the HwBinder stub (server-side) implementation of the interface. Developers should never refer to this class directly.
FooAll.cpp. A class that contains the implementations for both the HwBinder proxy and the HwBinder stub. When a client calls an interface method, the proxy automatically marshals the arguments from the client and sends the transaction to the binder kernel driver, which delivers the transaction to the stub on the other side (which then calls the actual server implementation).
The files are structured similarly to the files generated by aidl-cpp (for details, see "Passthrough mode" in the HIDL Overview). The only autogenerated file that is independent of the RPC mechanism used by HIDL is IFoo.h; all other files are tied to the HwBinder RPC mechanism used by HIDL. Therefore, client and server implementations should never directly refer to anything other than IFoo. To achieve this, include only IFoo.h and link against the generated shared library.

Note: HwBinder is only one possible transport; new transports might be added in the future.
Link to shared libraries
A client or server that uses any interface in a package must include the shared library of that package in one (1) of the following locations:

In Android.mk:

LOCAL_SHARED_LIBRARIES += android.hardware.samples@1.0
In Android.bp:

shared_libs: [
/* ... */
"android.hardware.samples@1.0",
],
Additional libraries you might need to include:

libhidlbase Includes standard HIDL data types. Starting in Android 10, this also contains all of the symbols previously in libhidltransport and libhwbinder.
libhidltransport Handles the transport of HIDL calls over different RPC/IPC mechanisms. Android 10 deprecates this library.
libhwbinder Binder-specific symbols. Android 10 deprecates this library.
libfmq Fast Message Queue IPC.
Namespaces
HIDL functions and types such as Return<T> and Void() are declared in namespace ::android::hardware. The C++ namespace of a package is determined by the package name and version. For example, a package mypackage with version 1.2 under hardware/interfaces has the following qualities:

C++ namespace is ::android::hardware::mypackage::V1_2
Fully qualified name of IMyInterface in that package is: ::android::hardware::mypackage::V1_2::IMyInterface. (IMyInterface is an identifier, not part of the namespace).
Types defined in the package's types.hal file are identified as: ::android::hardware::mypackage::V1_2::MyPackageType

Let’s break this down step-by-step and **explain in simple words** with examples:

🔹 What is this section about?

It explains how `.hal` files in **HIDL (HAL Interface Definition Language)** are:

- Structured into **packages**
- Compiled by **`hidl-gen`** into **C++ code**
- Turned into **shared libraries**
- Used by HAL clients and servers

🔸 Example `.hal` file

Suppose we create this HIDL file:

```hal
// hardware/interfaces/samples/1.0/IFoo.hal

package android.hardware.samples@1.0;

interface IFoo {
    struct Foo {
       int64_t someValue;
       handle  myHandle;
    };

    someMethod() generates (vec<uint32_t>);
    anotherMethod(Foo foo) generates (int32_t ret);
};
```

This defines an interface **`IFoo`** with two functions.

🔹 Where do we put `.hal` files?

Place your file like this:

```swift
hardware/interfaces/samples/1.0/IFoo.hal
```

This maps to the **package name**:

```css
android.hardware.samples@1.0
```

🔹 What happens when you compile using `hidl-gen`?

The **`hidl-gen`** tool reads your **`.hal`** file and **auto-generates many C++ files**, like:

File	Purpose
`IFoo.h`	The **main header** – defines the **`IFoo`** interface (used by both client and server).
`IHwFoo.h`	Internal serialization helpers (you don't use this).
`BpHwFoo.h`	Proxy (client-side code that sends RPC calls).
`BnHwFoo.h`	Stub (server-side code that receives RPC calls).
`FooAll.cpp`	Code that connects both proxy & stub, and does data marshaling.

✅ **You only use `IFoo.h` in your code** — everything else is internal.

🔹 What does the shared library look like?

All these files are compiled into a shared library:

```bash
android.hardware.samples@1.0.so
```

You link this **`.so`** in your **`Android.mk`** or **`Android.bp`**:

In Android.mk:

```makefile
LOCAL_SHARED_LIBRARIES += android.hardware.samples@1.0
```

In Android.bp:

```python
shared_libs: [
    "android.hardware.samples@1.0",
],
```

🔹 What are other libraries you might need?

Library	Use
`libhidlbase`	Basic HIDL support (required for clients and servers).
`libfmq`	Used if you're using Fast Message Queue in your **`.hal`**.
`libhidltransport` & **`libhwbinder`**	Older transport libs (mostly merged into **`libhidlbase`** in Android 10+).

🔹 What is the namespace in C++?

The package name gets converted to a **C++ namespace**.

For example:

If package is:

```css
android.hardware.samples@1.0
```

Then the C++ namespace is:

```cpp
::android::hardware::samples::V1_0
```

And you'll include the interface like:

```cpp
#include <android/hardware/samples/1.0/IFoo.h>
using android::hardware::samples::V1_0::IFoo;
```

🔹 Summary

Item	Meaning
`.hal` file	Defines the interface, like a header file
`hidl-gen`	Compiles `.hal` to `.h`/`.cpp` files
`IFoo.h`	Main file to include in client/server
`android.hardware.samples@1.0.so`	Shared library containing the compiled HAL
`libhidlbase`	Needed to use HIDL interfaces in C++
`namespace`	Based on package and version

🙋 Need a real-world demo?

I can show you a full `IFoo` HAL directory structure + client/server code setup. Just ask!

47. Interfaces

bookmark_border
Every interface defined in a HIDL package has its own autogenerated C++ class inside its package's namespace. Clients and servers deal with interfaces in different ways:

Servers implement interfaces.
Clients call methods on interfaces.
Interfaces can either be registered by name by the server or passed as parameters to HIDL-defined methods. For example, framework code can serve an interface to receive asynchronous messages from the HAL and pass that interface directly to the HAL without registering it.

Server implementation
A server implementing the IFoo interface must include the IFoo header file that was autogenerated:

```
#include <android/hardware/samples/1.0/IFoo.h>
```

The header is automatically exported by the shared library of the IFoo interface to link against. Example IFoo.hal:

```
// IFoo.hal
interface IFoo {
    someMethod() generates (vec<uint32_t>);
    ...
}
```

Example skeleton for a server implementation of the IFoo interface:

```
// From the IFoo.h header
using android::hardware::samples::V1_0::IFoo;

class FooImpl : public IFoo {
    Return<void> someMethod(foo my_foo, someMethod_cb _cb) {
        vec<uint32_t> return_data;
        // Compute return_data
        _cb(return_data);
        return Void();
    }
    ...
};
```

To make the implementation of a server interface available to a client, you can:

Register the interface implementation with the hwservicemanager (see details below),

OR

Pass the interface implementation as an argument of an interface method (for detals, see Asynchronous callbacks).
When registering the interface implementation, the hwservicemanager process keeps track of registered HIDL interfaces running on the device by name and version. Servers can register a HIDL interface implementation by name and clients can request service implementations by name and version. This process serves the HIDL interface android.hidl.manager@1.0::IServiceManager.

Each auto-generated HIDL interface header file (such as IFoo.h) has a registerAsService() method that can be used to register the interface implementation with the hwservicemanager. The only required argument is the name of the interface implementations as clients use this name to retrieve the interface from the hwservicemanager later:

```
::android::sp<IFoo> myFoo = new FooImpl();
::android::sp<IFoo> mySecondFoo = new FooAnotherImpl();
status_t status = myFoo->registerAsService();
status_t anotherStatus = mySecondFoo->registerAsService("another_foo");
```

The hwservicemanager treats the combination of [package@version::interface, instance_name] as unique to enable different interfaces (or different versions of the same interface) to register with identical instance names without conflicts. If you call registerAsService() with the exact same package version, interface, and instance name, the hwservicemanager drops its reference to the previously registered service and uses the new one.

Client implementation
Just as the server does, a client must #include every interface it refers to:

```
#include <android/hardware/samples/1.0/IFoo.h>
```

A client can obtain an interface in two ways:

Through I<InterfaceName>::getService (via the hwservicemanager)
Through an interface method
Each autogenerated interface header file has a static getService method that can be used to retrieve a service instance from the hwservicemanager:

```
// getService returns nullptr if the service can't be found
sp<IFoo> myFoo = IFoo::getService();
sp<IFoo> myAlternateFoo = IFoo::getService("another_foo");
```

Now the client has an an IFoo interface, and can call methods to it as if it were a local class implementation. In reality, the implementation can run in the same process, a different process, or even on another device (with HAL remoting). Because the client called getService on an IFoo object included from version 1.0 of the package, the hwservicemanager returns a server implementation only if that implementation is compatible with 1.0 clients. In practice, this means only server implementations with version 1.n (version x.(y+1) of an interface must extend (inherit from) x.y).

Additionally the method castFrom is provided to cast between different interfaces. This method works by making an IPC call to the remote interface to make sure the underlying type is the same as the type that is being requested. If the requested type is unavailable, then nullptr is returned.

```
sp<V1_0::IFoo> foo1_0 = V1_0::IFoo::getService();
sp<V1_1::IFoo> foo1_1 = V1_1::IFoo::castFrom(foo1_0);
```

Asynchronous callbacks
Many existing HAL implementations talk to asynchronous hardware, which means they need an asynchronous way to notify clients of new events that have occurred. A HIDL interface can be used as an asynchronous callback because HIDL interface functions can take HIDL interface objects as parameters.

Example interface file IFooCallback.hal:

```
package android.hardware.samples@1.0;
interface IFooCallback {
   sendEvent(uint32_t event_id);
   sendData(vec<uint8_t> data);
}
```

Example new method in IFoo that takes an IFooCallback parameter:

```
package android.hardware.samples@1.0;
interface IFoo {
   struct Foo {
      int64_t someValue;
      handle myHandle;
   };

   someMethod(Foo foo) generates (int32_t ret);
   anotherMethod() generates (vec<uint32_t>);
   registerCallback(IFooCallback callback);
};
```

The client using the IFoo interface is the server of the IFooCallback interface; it provides an implementation of IFooCallback:

```
class FooCallback : public IFooCallback {
   Return<void> sendEvent(uint32_t event_id) {
      // process the event from the HAL
   }
   Return<void> sendData(const hidl_vec<uint8_t>& data) {
      // process data from the HAL
   }
```

};
It can also simply pass that over an existing instance of the IFoo interface:

sp<IFooCallback> myFooCallback = new FooCallback();
myFoo.registerCallback(myFooCallback);
The server implementing IFoo receives this as an sp<IFooCallback> object. It can store the callback, and call back into the client whenever it wants to use this interface.

Death recipients
As service implementations can run in a different process, it can happen that the process implementing an interface dies while the client stays alive. Any calls on an interface object hosted in a process that has died fails with a transport error (isOK() returns false). The only way to recover from such a failure is to request a new instance of the service by calling I<InterfaceName>::getService(). This works only if the process that crashed has restarted and re-registered its services with the servicemanager (which is generally true for HAL implementations).

Instead of dealing with this reactively, clients of an interface can also register a death recipient to get a notification when a service dies. To register for such notifications on a retrieved IFoo interface, a client can do the following:

foo->linkToDeath(recipient, 1481 /* cookie */);
The recipient parameter must be an implementation of the android::hardware::hidl_death_recipient interface provided by HIDL, which contains a single method serviceDied() that's called from a thread in the RPC threadpool when the process hosting the interface dies:

class MyDeathRecipient : public android::hardware::hidl_death_recipient {
  virtual void serviceDied(uint64_t cookie, const android::wp<::android::hidl::base::V1_0::IBase>& who) {
    // Deal with the fact that the service died
  }
}
The cookie parameter contains the cookie that was passed in with linkToDeath(), whereas the who parameter contains a weak pointer to the object representing the service in the client. With the sample call given above, cookie equals 1481, and who equals foo.

It's also possible to unregister a death recipient after registering it:

foo->unlinkToDeath(recipient);

Let's break this down **step-by-step with simple explanations and examples** so you clearly understand how HIDL interfaces are implemented, used, and handled in C++.

🔹 1. What is a HIDL Interface?

A HIDL interface (e.g. `IFoo`) is like a **contract** between Android Framework and hardware vendors. It defines what **functions must be implemented** by a hardware module (like NFC, camera, etc.).

🔹 2. Two Roles Involved:

- **Server** = HAL vendor implementation (e.g., code in `vendor/`)
- **Client** = Framework or app that uses the HAL (e.g., system service)

🔹 3. Example Interface (`IFoo.hal`)

```hal
package android.hardware.samples@1.0;

interface IFoo {
    someMethod() generates (vec<uint32_t>);
}
```

🔹 4. Server-side (Vendor/HAL implementation)

✅ Include the generated header:

```cpp
#include <android/hardware/samples/1.0/IFoo.h>

using android::hardware::samples::V1_0::IFoo;
```

✅ Implement the interface:

```cpp
class FooImpl : public IFoo {
public:
    Return<void> someMethod(someMethod_cb _cb) override {
        // Create result
        hidl_vec<uint32_t> result = {1, 2, 3};
        _cb(result); // respond via callback
        return Void();
    }
};
```

✅ Register with `hwservicemanager`:

```cpp
sp<IFoo> myFoo = new FooImpl();
myFoo->registerAsService();  // default name = "default"
```

This exposes your implementation to clients via `hwservicemanager`.

🔹 5. Client-side (Framework/system service)

✅ Include interface:

```cpp
#include <android/hardware/samples/1.0/IFoo.h>
```

✅ Get the service:

```cpp
sp<IFoo> myFoo = IFoo::getService();  // Looks for "default"
```

✅ Call a method:

```cpp
myFoo->someMethod([](const hidl_vec<uint32_t>& vec) {
    for (auto val : vec) {
        ALOGI("value = %u", val);
    }
});
```

🔹 6. Asynchronous Callbacks (like event listeners)

Let's say the **HAL wants to notify the Framework** of events (like `onDataReceived()`).

✅ Define a callback interface:

```hal
// IFooCallback.hal
interface IFooCallback {
    sendEvent(uint32_t event_id);
}
```

✅ Modify IFoo:

```hal
interface IFoo {
    registerCallback(IFooCallback callback);
}
```

✅ Client provides the callback:

```cpp
class FooCallback : public IFooCallback {
public:
    Return<void> sendEvent(uint32_t event_id) override {
        ALOGI("Received event: %u", event_id);
        return Void();
    }
};

sp<IFooCallback> cb = new FooCallback();
myFoo->registerCallback(cb);  // Tell HAL how to notify
```

Now the **HAL can notify** the client via `cb->sendEvent(...)`.

🔹 7. Version Upgrade and Casting

HIDL supports **versioning**, e.g., `V1_0`, `V1_1`, etc.

Cast 1.0 object to 1.1:

```cpp
sp<V1_0::IFoo> foo1_0 = V1_0::IFoo::getService();
sp<V1_1::IFoo> foo1_1 = V1_1::IFoo::castFrom(foo1_0);

if (foo1_1 != nullptr) {
    foo1_1->newMethod(); // Only available in 1.1
}
```

🔹 8. Death Recipients (Detect if HAL process crashes)

If HAL crashes, your interface becomes **invalid**. You can detect this with a **death recipient**:

```cpp
class MyDeathRecipient : public hidl_death_recipient {
    void serviceDied(uint64_t cookie,
        const wp<IBase>& who) override {
        ALOGE("Service died. cookie: %llu", cookie);
    }
};

sp<MyDeathRecipient> recipient = new MyDeathRecipient();
myFoo->linkToDeath(recipient, 1234); // register
// ...
myFoo->unlinkToDeath(recipient); // unregister
```

Now if the HAL process crashes, `serviceDied()` is called.

✅ Summary Table

Concept	Server	Client
Interface	Implements `IFoo`	Calls `IFoo::getService()`
Callback	Receives and stores `IFooCallback`	Implements and passes `IFooCallback`
Registration	`registerAsService()`	`getService()`
Versioning	`@1.0`, `@1.1`	Use `castFrom()`
Crashes	HAL restarts itself	Use `linkToDeath()`

If you'd like, I can also show you:

- A full working client + server example using C++
- How to write `.hal`, compile it, and build into `.so`

Just say the word!

48. ↳
🔹 8. **Death Recipients (Detect if HAL process crashes)** -- If HAL crashes, your interface becomes **invalid**. You can detect this with a **death recipient**: cpp CopyEdit `class MyDeathRecipient : public hidl_death_recipient { void serviceDied(uint64_t cookie, const wp<IBase>& who) override { ALOGE("Service died. cookie: %llu", cookie); } }; sp<MyDeathRecipient> recipient = new MyDeathRecipient(); myFoo->linkToDeath(recipient, 1234); // register // ... myFoo->unlinkToDeath(recipient); // unregister`
explain the code here

Sure, let's break down this **death recipient** code in **very simple terms**, line by line.

🔹 Purpose of This Code

The **death recipient** lets the **client** know if the **HAL (server)** crashes (dies).
This is useful because once the server dies, any call to its methods will fail. You can then try to recover (e.g., by getting the service again).

🔸 Code Breakdown

```cpp
class MyDeathRecipient : public hidl_death_recipient {
```

- You're defining a **class** called **`MyDeathRecipient`**.
- It **inherits** from **`hidl_death_recipient`**, which is a base class provided by HIDL.
- This base class lets you receive a **callback** when the HAL dies.

```cpp
    void serviceDied(uint64_t cookie, const wp<IBase>& who) override {
```

- **`serviceDied(...)`** is the function that's automatically called when the HAL process crashes.
- **`cookie`**: Just a number that helps you know **which** service died (you pass it when you register).
- **`who`**: A **weak pointer** to the object that died (so you know **which interface** is dead).

```cpp
        ALOGE("Service died. cookie: %llu", cookie);
    }
};
```

- Inside **`serviceDied()`**, you're just printing a log saying the service died.
- **`ALOGE`** is Android's macro to print error logs.

✅ Register the death recipient:

```cpp
sp<MyDeathRecipient> recipient = new MyDeathRecipient();
```

- You create an instance of your death recipient.

```cpp
myFoo->linkToDeath(recipient, 1234);
```

- You link the death recipient to the **`myFoo`** service.
- Now, if **`myFoo`** (the HAL) crashes, your **`serviceDied()`** function will be called.

- **1234** is the **cookie** you use to identify this particular registration.

🧹 Unregister the death recipient:

```cpp
myFoo->unlinkToDeath(recipient);
```

- This removes the link. After this, your **serviceDied()** won't be called anymore.

📌 Example Scenario

1. You get a service:

   ```cpp
   sp<IFoo> myFoo = IFoo::getService();
   ```

2. You link to death:

   ```cpp
   myFoo->linkToDeath(recipient, 1234);
   ```

3. Suddenly, **myFoo**'s server process crashes.
4. Then your code automatically receives:

   ```cpp
   serviceDied(1234, myFoo);
   ```

5. You can then:
   - Try to call **IFoo::getService()** again
   - Show a UI error
   - Retry logic, etc.

Would you like me to write a complete working example using **IFoo** and **MyDeathRecipient**?

49. yes

Great! Below is a **complete working example** showing how a **HIDL client**:

- Gets a service **`IFoo`**
- Links a **death recipient**
- Handles the case when the HAL **dies**

✅ Assumptions

- You have a HIDL interface: **`IFoo.hal`**
- The package is: **`android.hardware.samples@1.0`**

🧱 HIDL Interface

Let's say **`IFoo.hal`** looks like this:

```hidl
package android.hardware.samples@1.0;

interface IFoo {
    someMethod() generates (int32_t ret);
};
```

✅ C++ Client Example (using `IFoo`)

```cpp
#include <android/hardware/samples/1.0/IFoo.h>
#include <hidl/HidlSupport.h>
#include <hidl/HidlTransportSupport.h>
#include <log/log.h>

using android::hardware::samples::V1_0::IFoo;
using android::sp;
using android::wp;
using android::hardware::hidl_death_recipient;
using android::hardware::Return;
using android::hardware::Void;

// 1. Death recipient class
class MyDeathRecipient : public hidl_death_recipient {
public:
    virtual void serviceDied(uint64_t cookie, const wp<::android::hidl::base::V1_0::IBase>& who) override {
        ALOGE("Service died! Cookie: %llu", static_cast<unsigned long long>(cookie));
        // You can now attempt recovery here (e.g., getService() again)
    }
};

int main() {
    // 2. Get the HIDL service
    sp<IFoo> foo = IFoo::getService();
    if (foo == nullptr) {
        ALOGE("Failed to get IFoo service!");
        return 1;
    }

    // 3. Set up death recipient
    sp<MyDeathRecipient> deathRecipient = new MyDeathRecipient();
    if (!foo->linkToDeath(deathRecipient, 1234 /*cookie*/)) {
        ALOGE("Failed to link death recipient");
        return 1;
    }

    ALOGI("IFoo service is up and running");

    // 4. Use the service
    Return<int32_t> result = foo->someMethod();
    if (!result.isOk()) {
        ALOGE("Call failed, transport error");
        return 1;
```

```
    }
    ALOGI("someMethod returned: %d", static_cast<int>(result));

    // Simulate work
    sleep(10);

    // 5. Optional: Unlink when done
    foo->unlinkToDeath(deathRecipient);

    return 0;
}
```

🔄 What Happens at Runtime

- **`getService()`** connects to the HAL.
- If the HAL crashes (or is killed), your **`MyDeathRecipient::serviceDied()`** is called.
- You can then log, retry, or restart your service logic.

📌 Notes

- You must **`#include`** the **`.hal`**-generated **`IFoo.h`** header.
- Use **`ALOGE`**/**`ALOGI`** macros to print to logcat.
- Make sure to link the correct **`.so`** libraries in **`Android.mk`** or **`Android.bp`**:

```make
make

LOCAL_SHARED_LIBRARIES += android.hardware.samples@1.0 libhidlbase liblog
```

Would you like a **binderized service code** too for **`IFoo`**, so you can test this client?

50. ↪
sp<MyDeathRecipient> deathRecipient = new MyDeathRecipient(); if (!foo->linkToDeath(deathRecipient, 1234 /*cookie*/)) { ALOGE("Failed to link death recipient"); return 1; }
explain the syntax

Sure! Let's break down this line-by-line in **very simple terms**, explaining the **syntax** and **meaning** of each part of this HIDL death recipient setup:

🔹 Code:

```cpp
sp<MyDeathRecipient> deathRecipient = new MyDeathRecipient();
```

🔍 Meaning:

- `sp<...>`: A **strong pointer** from Android's HIDL system (`android::sp`). It behaves like a smart pointer (reference counted).
- `MyDeathRecipient`: This is your **custom class** that handles the `serviceDied()` callback.
- `new MyDeathRecipient()`: Creates an **object** of your death recipient on the heap.

➡️ This line means:
"Create a smart pointer to a death recipient object that we can register for callbacks."

🔹 Code:

```cpp
if (!foo->linkToDeath(deathRecipient, 1234 /*cookie*/)) {
```

🔍 Meaning:

- `foo`: This is your interface to the HAL (e.g., `IFoo::getService()`).
- `linkToDeath(...)`: A method provided by HIDL to register for death notifications.
- `deathRecipient`: The object you want to be notified if `foo` dies.
- `1234`: A number called a **cookie** (just an identifier to help you recognize which service died if multiple are linked).

➡️ This line means:
"Ask the HAL `foo` to notify `deathRecipient` if it crashes or dies. Use `1234` to identify this particular link."

🔹 Code:

```cpp
    ALOGE("Failed to link death recipient");
    return 1;
```

🔍 Meaning:

- `ALOGE(...)`: Print an error to **logcat** if linking the death recipient failed.
- `return 1`: Exit the program with error code `1`.

➡️ This block means:
"If linking the death callback fails, log the error and stop the program."

✅ Summary (in Plain English)

> "Create a death listener. Then tell the HAL: *If you crash, tell this listener*. If setting up the listener fails, report an error and quit."

Let me know if you'd like me to walk through what happens **when the HAL dies** and how `serviceDied()` is triggered!

51. Data types

bookmark_border
HIDL data declarations generate C++ standard-layout data structures. These structures can be placed anywhere that feels natural (on the stack, at file or global scope, or on the heap) and can be composed in the same fashion. Client code calls HIDL proxy code passing in const references and primitive types, while the stub and proxy code hides the details of serialization.

Note: At no point is developer-written code required to explicitly serialize or deserialize data structures.

The table below maps HIDL primitives to C++ data types:

HIDL type	C++ Type	Header/library
enum	enum class	
uint8_t..uint64_t	uint8_t..uint64_t	<stdint.h>
int8_t..int64_t	int8_t..int64_t	<stdint.h>
float	float	
double	double	
vec<T>	hidl_vec<T>	libhidlbase
T[S1][S2]...[SN]	T[S1][S2]...[SN]	
string	hidl_string	libhidlbase
handle	hidl_handle	libhidlbase
safe_union	(custom) struct	
struct	struct	
union	union	
fmq_sync	MQDescriptorSync	libhidlbase
fmq_unsync	MQDescriptorUnsync	libhidlbase

The sections below describe data types in more detail.

enum
An enum in HIDL becomes an enum in C++. For example:

```
enum Mode : uint8_t { WRITE = 1 << 0, READ = 1 << 1 };
enum SpecialMode : Mode { NONE = 0, COMPARE = 1 << 2 };
```

... becomes:

```
enum class Mode : uint8_t { WRITE = 1, READ = 2 };
enum class SpecialMode : uint8_t { WRITE = 1, READ = 2, NONE = 0, COMPARE = 4 };
```

Starting in Android 10, an enum can be iterated over using ::android::hardware::hidl_enum_range. This range includes every enumerator in the order it appears in HIDL source code, starting from the parent enum down to the last child. For example, this code iterates over WRITE, READ, NONE, and COMPARE in that order. Given SpecialMode above:

```
template <typename T>
using hidl_enum_range = ::android::hardware::hidl_enum_range<T>

for (SpecialMode mode : hidl_enum_range<SpecialMode>) {...}
```

hidl_enum_range also implements reverse iterators and can be used in constexpr contexts. If a value appears in an enumeration multiple times, the value appears in the range multiple times.

bitfield<T>
bitfield<T> (where T is a user-defined enum) becomes the underlying type of that enum in C++. In the above example, bitfield<Mode> becomes uint8_t.

vec<T>
The hidl_vec<T> class template is part of libhidlbase and can be used to pass a vector of any HIDL type with an arbitrary size. The comparable fixed size container is hidl_array. A hidl_vec<T> can also be initialized to point to an external data buffer of type T, using the hidl_vec::setToExternal() function.

In addition to emitting/inserting the struct appropriately in the generated C++ header, the use of vec<T> generates some convenience functions to translate to/from std::vector and bare T pointers. If the vec<T> is used as a parameter, the function using it's overloaded (two prototypes are generated) to accept and pass both the HIDL struct and a std::vector<T> type for that parameter.

array
Constant arrays in hidl are represented by the hidl_array class in libhidlbase. A hidl_array<T, S1, S2, …, SN> represents an N dimensional fixed size array T[S1][S2]…[SN].

string
The hidl_string class (part of libhidlbase) can be used to pass strings over HIDL interfaces and is defined in /system/libhidl/base/include/hidl/HidlSupport.h. The first storage location in the class is a pointer to its character buffer.

hidl_string knows how to convert to and from std::string and char* (C-style string) using operator=, implicit casts, and .c_str() function. HIDL string structs has the appropriate copy constructors and assignment operators to:

Load the HIDL string from an std::string or a C string.
Create a new std::string from a HIDL string.
In addition, HIDL strings have conversion constructors so C strings (char *) and C++ strings (std::string) can be used on methods that take a HIDL string.

struct
A struct in HIDL can contain only fixed-size data types and no functions. HIDL struct definitions map directly to standard-layout structs in C++, ensuring that structs have a consistent memory layout. A struct can include HIDL types, including handle, string, and vec<T>, that point to separate variable-length buffers.

handle
WARNING: Addresses of any kind (even physical device addresses) must never be part of a native handle. Passing this information between processes is dangerous and makes them susceptible to attack. Any values passed between processes must be validated before being used to look up allocated memory within a process. Otherwise, bad handles can cause bad memory access or memory corruption.

The handle type is represented by the hidl_handle structure in C++, which is a simple wrapper around a pointer to a const native_handle_t object (this has been present in Android for a long time).

```
typedef struct native_handle
{
    int version;        /* sizeof(native_handle_t) */
    int numFds;         /* number of file descriptors at &data[0] */
    int numInts;        /* number of ints at &data[numFds] */
    int data[0];        /* numFds + numInts ints */
} native_handle_t;
```
By default, hidl_handle does not take ownership of the native_handle_t pointer it wraps. It merely exists to safely store a pointer to a native_handle_t such that it can be used in both 32- and 64-bit processes.

Scenarios in which the hidl_handle does own its enclosed file descriptors include:

Following a call to the setTo(native_handle_t* handle, bool shouldOwn) method with the shouldOwn parameter set to true
When the hidl_handle object is created by copy construction from another hidl_handle object
When the hidl_handle object is copy-assigned from another hidl_handle object
hidl_handle provides both implicit and explicit conversions to/from native_handle_t* objects. The main use for the handle type in HIDL is to pass file descriptors over HIDL interfaces. A single file descriptor is therefore represented by a native_handle_t with no ints and a single fd. If the client and server live in a different process, the RPC implementation automatically takes care of the file descriptor to ensure both processes can operate on the same file.

Although a file descriptor received in a hidl_handle by a process is valid in that process, it doesn't persist beyond the receiving function (it's closed when the function returns). A process that wants to retain persistent access to the file descriptor must dup() the enclosed file descriptors, or copy the entire hidl_handle object.

memory
The HIDL memory type maps to the hidl_memory class in libhidlbase, which represents unmapped shared memory. This is the object that must be passed between processes to share memory in HIDL. To use shared memory:

Obtain an instance of IAllocator (currently only instance "ashmem" is available) and use it to allocate shared memory.
IAllocator::allocate() returns a hidl_memory object that can be passed through HIDL RPC and be mapped into a process using libhidlmemory's mapMemory function.
mapMemory returns a reference to an sp<IMemory> object that can be used to access the memory. (IMemory and IAllocator are defined in android.hidl.memory@1.0.)
An instance of IAllocator can be used to allocate memory:

#include <android/hidl/allocator/1.0/IAllocator.h>
#include <android/hidl/memory/1.0/IMemory.h>
#include <hidlmemory/mapping.h>
using ::android::hidl::allocator::V1_0::IAllocator;
using ::android::hidl::memory::V1_0::IMemory;
using ::android::hardware::hidl_memory;
....
 sp<IAllocator> ashmemAllocator = IAllocator::getService("ashmem");
 ashmemAllocator->allocate(2048, [&](bool success, const hidl_memory& mem) {
 if (!success) { /* error */ }
 // now you can use the hidl_memory object 'mem' or pass it around
 }));
Actual changes to the memory must be done through an IMemory object, either on the side that created mem or on the side that receives it over HIDL RPC.

// Same includes as above

sp<IMemory> memory = mapMemory(mem);
void* data = memory->getPointer();
memory->update();
// update memory however you wish after calling update and before calling commit
data[0] = 42;
memory->commit();
// ...
memory->update(); // the same memory can be updated multiple times
// ...
memory->commit();
interface
Interfaces can be passed as objects. The word interface can be used as syntactic sugar for the type android.hidl.base@1.0::IBase; in addition, the current interface and any imported interfaces are defined as a type.

Variables that hold Interfaces should be strong pointers: sp<IName>. HIDL functions that take interface parameters convert raw pointers to strong pointers, causing nonintuitive behavior (the pointer can be cleared unexpectedly). To avoid problems, always store HIDL interfaces as a sp<>.

Great — you've gone through a very detailed section on **HIDL data types** and how they map to **C++ equivalents**. Here's a simple breakdown to help reinforce it with examples:

🔧 Common HIDL → C++ Type Mappings

HIDL Type	C++ Type	Used For
`int32_t`	`int32_t`	Regular 32-bit integers
`string`	`hidl_string`	Dynamic text strings
`vec<int32_t>`	`hidl_vec<int32_t>`	Dynamic arrays like `std::vector`
`int32_t[4]`	`int32_t[4]`	Fixed-size arrays
`handle`	`hidl_handle`	File descriptors
`memory`	`hidl_memory`	Shared memory
`enum`	`enum class`	Enumerations with fixed values
`interface`	`sp<IMyInterface>`	A reference-counted interface obj

📦 Example: Define a Struct and Use in Interface

`IFoo.hal`

```hidl
package android.hardware.example@1.0;

interface IFoo {
    struct Point {
        int32_t x;
```

```
        int32_t y;
    };

    drawPoint(Point pt);
};
```

This will generate:

`IFoo.h` (auto-generated C++)

```cpp
namespace android::hardware::example::V1_0 {

struct IFoo : public ::android::hidl::base::V1_0::IBase {
    struct Point {
        int32_t x;
        int32_t y;
    };

    virtual Return<void> drawPoint(const Point& pt) = 0;
};
}
```

🧠 What's `hidl_vec<T>`?

Equivalent to `std::vector<T>`, used like:

```cpp
hidl_vec<int32_t> values;
values.resize(3);
values[0] = 10;
values[1] = 20;
values[2] = 30;
```

Or initialize from a `std::vector`:

```cpp
std::vector<int32_t> vec = {1, 2, 3};
hidl_vec<int32_t> hvec(vec);
```

🧵 What's `hidl_string`?

Dynamic string compatible with `std::string`:

```cpp
hidl_string hstr = "Hello World";
std::string stdstr = hstr; // implicit conversion
```

💾 What's `hidl_handle` for?

Used to pass file descriptors safely (e.g., a shared memory fd). You never access its internal data directly. It's mainly passed to the kernel or HALs.

💡 What's `hidl_memory` and `IMemory`?

They allow passing and accessing shared memory:

```cpp
sp<IAllocator> allocator = IAllocator::getService("ashmem");
allocator->allocate(4096, [&](bool success, const hidl_memory& mem) {
    sp<IMemory> memory = mapMemory(mem);
    void* data = memory->getPointer();
    memory->update();
    ((int*)data)[0] = 42;
    memory->commit();
});
```

Would you like a working server+client code example using `vec`, `string`, or `memory`?

52. Functions

bookmark_border
Functions in a HIDL interface are mapped to methods in the autogenerated IFoo C++ class declaration. The name of each function remains the same in C++; the following sections describe how HIDL arguments and return values are translated to C++.

Function parameters
The arguments listed in the .hal file map to C++ data types. Arguments that do not map to a primitive C++ type are passed by const reference.

For every HIDL function that has a return value (has a generates statement), the C++ parameter list for that function has an additional argument: a callback function that is called with the return values of the HIDL function. There is one exception: If the generates clause contains a single parameter that directly maps to a C++ primitive, callback elision is used (the callback is removed and the return value is returned from the function through a normal return statement).

Function return values
The following functions have return values.

Transport errors and return type
The generates statement can result in three types of function signatures:

For only one return value that is a C++ primitive, the generates return value is returned by value from the function in a Return<T> object.
For more complicated cases, the generates return value(s) are returned through the callback parameter provided with the function call itself, and the function returns Return<void>.
For when no generates statement exists, the function returns Return<void>.
RPC calls can occasionally encounter transport errors, e.g. when the server dies, when transport resources are insufficient to complete the call, or when the parameters passed do not permit completing the call (such as missing a required callback function). Return objects store transport error indications as well as a T value (except Return<void>).

As the client-side and server-side functions have the same signature, the server-side function must return a Return type even though its implementation does not signal transport errors. Return<T> objects are constructed with Return(myTValue) (or can be implicitly constructed from mTValue, such as in return statements) and Return<void> objects are constructed with Void().

Return<T> objects have implicit conversion to and from their T value. The Return object can be checked for transport errors by calling its isOk() method. This check is not required; however, if an error occurs and is not checked by the time the Return object is destroyed, or a T value conversion is attempted, the client process will be killed and an error logged. If isOk() indicates a transport error or a call failure due to a logic error in developer code (such as passing nullptr as a synchronous callback), then description() can be called on the Return object to return a string suitable for logging. In such cases, there is no way to determine how much code may have executed on the server as a result of the failed call. The method isDeadObject() is also provided. This method indicates that !isOk() is because the remote object has crashed or no longer exists. isDeadObject() always implies !isOk().

Return by value
If the generates statement maps to a single C++ primitive, no callback parameter is in the parameter list. Instead, an implementation provides the return value T in a Return<T> object, which can be implicitly generated from the primitive type T. For example:

```
Return<uint32_t> someMethod() {
    uint32_t return_data = ...; // Compute return_data
    return return_data;
};
```

The method Return<*>::withDefault is also provided. This method provides a value in cases where the return value is !isOk(). This method also automatically marks the return object as okay so the client process will not be killed.

Return using callback parameter
A callback can pass the return value of the HIDL function back to the caller. The prototype of the callback is a std::function object with parameters (taken from the generates statement) mapped to C++ types. Its return value is void—the callback itself doesn't return a value.

The return value of a C++ function with a callback parameter has type Return<void>. The server implementation is responsible only for providing the return value. As the return values are already transferred using the callback, the T template parameter is void:

```
Return<void> someMethod(someMethod_cb _cb);
```

From their C++ implementation, server implementations should return Void(), which is a static inlined function returning a Return<void> object.
Example of a typical server method implementation with a callback parameter:

Return<void> someMethod(someMethod_cb _cb) {
// Do some processing, then call callback with return data
hidl_vec<uint32_t> vec = ...
_cb(vec);
return Void();
};
Functions without return values
The C++ signature of a function without a generates statement will not have a callback parameter in the parameter list. Its return type will be Return<void>.

Oneway functions
Functions marked with the oneway keyword are asynchronous functions (clients won't block on their execution) and do not have return values. The C++ signature of a oneway function will not have a callback parameter in the parameter list, and its C++ return value will be Return<void>.

Let's break this section down **step by step**, with **simple examples** so you clearly understand how **HIDL functions** behave and how they map to **C++**.

🧠 Basics: HIDL → C++ Function Mapping

In HIDL, functions are written in `.hal` files like this:

```hidl
interface IExample {
    doSomething();
    getValue() generates (uint32_t val);
    process(vec<uint8_t> data) generates (bool success);
}
```

These are converted into C++ method signatures automatically.

✅ Function Types and Their C++ Mappings

1. Function with No Return (`doSomething`)

- HIDL: `doSomething();`
- C++:

```cpp
Return<void> doSomething();
```

This is a **simple method**, does not return any value.

---

2. Single Primitive Return (`getValue`)

- HIDL: `getValue() generates (uint32_t val);`
- C++:

```cpp
Return<uint32_t> getValue();
```

✔️ No callback needed here because it's a **single primitive type**.

Usage in C++:

```cpp
Return<uint32_t> ret = iface->getValue();
if (ret.isOk()) {
    uint32_t value = ret;
}
```

3. Multiple or Complex Return Values (`process`)

- HIDL: **`process(vec<uint8_t> data) generates (bool success);`**
- C++:

```cpp
Return<void> process(const hidl_vec<uint8_t>& data, process_cb _cb);
```

⚠️ This uses a **callback** to return values.

Server-side example:

```cpp
Return<void> process(const hidl_vec<uint8_t>& data, process_cb _cb) {
    bool success = data.size() > 0;
    _cb(success);  // Return via callback
    return Void(); // Required return type
}
```

Client-side:

```cpp
iface->process(myData, [](bool success) {
    if (success) {
        LOG(INFO) << "Processing successful!";
    }
});
```

💬 Callback Elision

When return is **just one primitive**, HIDL skips the callback for simplicity.

HIDL:

```hidl
getStatus() generates (bool status);
```

C++:

```cpp
Return<bool> getStatus();
```

No need to do:

```cpp
getStatus([](bool status){});
```

🔄 Transport Errors Handling

Any HIDL call returns a **`Return<T>`** or **`Return<void>`**, which can be checked:

```cpp
auto ret = iface->getValue();
if (!ret.isOk()) {
    LOG(ERROR) << "Binder failed: " << ret.description();
}
```

🧵 Oneway Functions (Asynchronous)

HIDL:

```hidl
oneway notifyEvent();
```

C++:

```cpp
Return<void> notifyEvent();
```

- Executes **asynchronously** (non-blocking).
- Can't return anything.
- Use when you don't expect a response.

🧩 Summary Table

HIDL Function Type	C++ Signature	Return Type	Uses Callback?
`doSomething()`	`Return<void> doSomething();`	`Return<void>`	❌
`getValue() generates (uint32_t)`	`Return<uint32_t> getValue();`	`Return<uint32_t>`	❌
`compute() generates (vec<int>)`	`Return<void> compute(compute_cb _cb);`	`Return<void>`	✅
`oneway notifyEvent()`	`Return<void> notifyEvent();`	`Return<void>`	❌

Would you like me to show how this would look in a full client-server implementation with a sample `.hal` file and C++ code?

53. ↪
Single Primitive Return (`getValue`) * HIDL: `getValue() generates (uint32_t val);` * C++: cpp CopyEdit `Return<uint32_t> getValue();` ✔️ No callback needed here because it's a **single primitive type**. Usage in C++: cpp CopyEdit `Return<uint32_t> ret = iface->getValue(); if (ret.isOk()) { uint32_t value = ret; }`
confusing

Great — let's simplify that explanation.

🎯 What's happening in this function?

Suppose your HIDL interface is:

```hidl
interface IExample {
    getValue() generates (uint32_t val);
}
```

This means:
📦 **Client calls `getValue()`**
✅ **It returns a number** (`uint32_t`) back to the client.

✅ What does the C++ code look like?

HIDL compiler converts it to:

```cpp
Return<uint32_t> getValue();
```

So, when the **client** calls this:

```cpp
sp<IExample> iface = IExample::getService();
Return<uint32_t> result = iface->getValue();
```

🤔 What's this `Return<uint32_t>` thing?

It's **not** just `uint32_t`. It's a wrapper that:

- ✅ Contains your actual result (like `42`)
- ⚠️ Also tracks transport errors (e.g., server crashed)

So, you must check:

```cpp
if (result.isOk()) {
    uint32_t value = result;  // Automatically extracts value
    LOG(INFO) << "Received: " << value;
} else {
    LOG(ERROR) << "HIDL call failed: " << result.description();
}
```

> ☑️ **`result.isOk()`** = no error
> 🔁 **`uint32_t value = result;`** = extract actual number

💡 Why no callback here?

Because `uint32_t` is a simple, primitive type.
➡️ HIDL optimizes it by **not using a callback** — it just returns the value directly inside `Return<T>`.

But if it had **multiple values** or **complex types**, a callback would be required.

✅ Summary

HIDL	C++	Notes
`getValue() generates (uint32_t);`	`Return<uint32_t> getValue();`	Simple return, no callback
Client usage	`uint32_t val = result;`	Only after checking `.isOk()`

Would it help if I gave you a **full working server + client example** for this case?

54. when will be callback and when not

Great question!

The rule for **when HIDL uses a callback vs not** depends on **how many values are returned** and **what kind of types they are**.

✅ When NO callback is used (direct return):

Use **direct return** (i.e., `Return<T>`) when:

- Your HIDL `generates` only **one** return value
- And that value is a **simple C++ primitive**:
  - `int32_t`, `uint64_t`, `bool`, etc.

Example HIDL:

```hidl
getCount() generates (int32_t count);
```

C++:

```cpp
Return<int32_t> getCount();
```

No callback needed.

🔁 When CALLBACK is used:

A **callback is used** when:

- There are **multiple** return values
- OR the return type is **not a simple C++ primitive**, like:
  - `vec<T>` → `hidl_vec<T>`
  - `string` → `hidl_string`
  - `struct`
  - `handle`, `memory`, etc.

Example HIDL (multiple values):

```hidl
getInfo() generates (string name, int32_t age);
```

C++:

```cpp
Return<void> getInfo(getInfo_cb _cb);
```

Then in your server:

```cpp
Return<void> getInfo(getInfo_cb _cb) {
    _cb("Sanjay", 27);
    return Void();
}
```

Example HIDL (complex type):

```hidl
getList() generates (vec<uint32_t> items);
```

C++:

```cpp
Return<void> getList(getList_cb _cb);
```

Then in your server:

```cpp
Return<void> getList(getList_cb _cb) {
    hidl_vec<uint32_t> items = {1, 2, 3, 4};
    _cb(items);
    return Void();
}
```

⚠️ Summary:

HIDL generates	Uses Callback?	C++ Signature Example
`generates (int32_t)`	❌ No	`Return<int32_t> method();`
`generates (string)`	✅ Yes	`Return<void> method(method_cb _cb);`
`generates (vec<int>)`	✅ Yes	`Return<void> method(method_cb _cb);`
`generates (a, b)`	✅ Yes	`Return<void> method(method_cb _cb);`
No return	❌ No	`Return<void> method();`

Let me know if you want a **real code example** for each!

55. HIDL Java

bookmark_border
In Android 8.0 the Android OS was re-architected to define clear interfaces between the device-independent Android platform, and device- and vendor-specific code. Android already defined many such interfaces in the form of HAL interfaces, defined as C headers in hardware/libhardware. HIDL replaced these HAL interfaces with stable, versioned interfaces, which can either be in Java (described below) or be client- and server-side HIDL interfaces in C++.

HIDL interfaces are intended to be used primarily from native code, and as a result HIDL is focused on the auto-generation of efficient code in C++. However, HIDL interfaces must also be available for use directly from Java, as some Android subsystems (such as Telephony) have Java HIDL interfaces.

The pages in this section describe the Java frontend for HIDL interfaces, detail how to create, register, and use services, and explain how HALs and HAL clients written in Java interact with the HIDL RPC system.

Client example
This is an example of a client for an interface IFoo in package android.hardware.foo@1.0 that is registered as service name default and an additional service with the custom service name second_impl.

Add libraries
You need to add dependencies on the corresponding HIDL stub library if you want to use it. Usually, this is a static library:

```
// in Android.bp
static_libs: [ "android.hardware.foo-V1.0-java", ],
// in Android.mk
LOCAL_STATIC_JAVA_LIBRARIES += android.hardware.foo-V1.0-java
```

If you know that you're already pulling in dependencies on these libraries, you can also use shared linkage:

```
// in Android.bp
libs: [ "android.hardware.foo-V1.0-java", ],
// in Android.mk
LOCAL_JAVA_LIBRARIES += android.hardware.foo-V1.0-java
```

Additional considerations for adding libraries in Android 10
If you have a system or vendor app that targets Android 10 or higher, you can statically include these libraries. You can also use (only) HIDL classes from custom JARs installed on the device with stable Java APIs made available using the existing uses-library mechanism for system apps. The latter approach saves space on the device. For more details, see Implementing Java SDK Library. For older apps, the old behavior is preserved.

Starting in Android 10, "shallow" versions of these libraries are also available. These include the class in question but don't include any of the dependent classes. For instance, android.hardware.foo-V1.0-java-shallow includes classes in the foo package, but doesn't include classes in android.hidl.base-V1.0-java, which contains the base class of all HIDL interfaces. If you're creating a library that already has the preferred interface's base classes available as a dependency, you can use the following:

```
// in Android.bp
static_libs: [ "android.hardware.foo-V1.0-java-shallow", ],
// in Android.mk
LOCAL_STATIC_JAVA_LIBRARIES += android.hardware.foo-V1.0-java-shallow
```

HIDL base and manager libraries are also no longer available on the boot classpath for apps (previously, they were sometimes used as hidden API, due to Android's delegate-first classloader). Instead, they've been moved into a new namespace with jarjar, and apps which use these (necessarily priv apps) must have their own separate copies. Modules on the boot classpath using HIDL must use the shallow variants of these Java libraries and to add jarjar_rules: ":framework-jarjar-rules" to their Android.bp to use the version of these libraries that exists in the boot classpath.

Modify your Java source
There's only one version (@1.0) of this service, so this code retrieves only that version. See interface extensions for how to handle multiple different versions of the service.

```
import android.hardware.foo.V1_0.IFoo;
...
// retry to wait until the service starts up if it is in the manifest
```

IFoo server = IFoo.getService(true /* retry */); // throws NoSuchElementException if not available
IFoo anotherServer = IFoo.getService("second_impl", true /* retry */);
server.doSomething(…);
Warning: Java getService with no arguments doesn't wait for the service to start.
Provide a service
Framework code in Java might need to serve interfaces to receive asynchronous callbacks from HALs.

Warning: Don't implement a driver (HAL) in Java. Google recommends that you implement drivers in C++.
Warning: Java drivers must be in a separate process from their clients (same process communication isn't supported).
For the IFooCallback interface in version 1.0 of the android.hardware.foo package, you can implement your interface in Java using the following steps:

Define your interface in HIDL.
Open /tmp/android/hardware/foo/IFooCallback.java as a reference.
Create a new module for your Java implementation.
Examine the abstract class android.hardware.foo.V1_0.IFooCallback.Stub, then write a new class to extend it and implement the abstract methods.
View auto-generated files
To view the automatically generated files, run:

```
hidl-gen -o /tmp -Ljava \
  -randroid.hardware:hardware/interfaces \
  -randroid.hidl:system/libhidl/transport android.hardware.foo@1.0
```

These commands generate the directory /tmp/android/hardware/foo/1.0. For the file hardware/interfaces/foo/1.0/IFooCallback.hal, this generates the file /tmp/android/hardware/foo/1.0/IFooCallback.java, which encapsulates the Java interface, the proxy code, and the stubs (both proxy and stubs conform to the interface).

-Lmakefile generates the rules that run this command at build time and allow you to include android.hardware.foo-V1.0-java and link against the appropriate files. A script that automatically does this for a project full of interfaces can be found at hardware/interfaces/update-makefiles.sh. The paths in this example are relative; hardware/interfaces can be a temporary directory under your code tree to enable you to develop a HAL prior to publishing it.

Run a service
The HAL provides the IFoo interface, which must make asynchronous callbacks to the framework over the IFooCallback interface. The IFooCallback interface isn't registered by name as a discoverable service; instead, IFoo must contain a method such as setFooCallback(IFooCallback x).

To set up IFooCallback from version 1.0 of the android.hardware.foo package, add android.hardware.foo-V1.0-java to Android.mk. The code to run the service is:

```
import android.hardware.foo.V1_0.IFoo;
import android.hardware.foo.V1_0.IFooCallback.Stub;
....
class FooCallback extends IFooCallback.Stub {
    // implement methods
}
....
// Get the service from which you will be receiving callbacks.
// This also starts the threadpool for your callback service.
IFoo server = IFoo.getService(true /* retry */); // throws NoSuchElementException if not available
....
// This must be a persistent instance variable, not local,
//   to avoid premature garbage collection.
FooCallback mFooCallback = new FooCallback();
....
// Do this once to create the callback service and tell the "foo-bar" service
server.setFooCallback(mFooCallback);
```

Interface extensions
Assuming a given service implements the IFoo interface across all devices, it's possible that on a particular device the service might provide additional capabilities implemented in the interface extension IBetterFoo, as follows:

```
interface IFoo {
   ...
};
```

```
interface IBetterFoo extends IFoo {
  ...
};
Calling code aware of the extended interface can use the castFrom() Java method to safely cast the base interface to the extended interface:


IFoo baseService = IFoo.getService(true /* retry */); // throws NoSuchElementException if not available
IBetterFoo extendedService = IBetterFoo.castFrom(baseService);
if (extendedService != null) {
  // The service implements the extended interface.
} else {
  // The service implements only the base interface.
}
```

Let's break this **HIDL Java** explanation into **simple, structured concepts** with **examples**, so it's easier to understand.

🧩 What is HIDL Java?

HIDL (HAL Interface Definition Language) is mostly used in **C++**, but some **Android subsystems like Telephony** are written in **Java**.

So, HIDL also supports **Java frontends**, allowing HAL interfaces to be accessed or implemented in Java.

✅ What You Can Do in Java (and what not)

Action	Java Support
Use HIDL interface (client)	✅ Yes
Implement HIDL interface (server)	⚠️ Yes, but limited
Implement actual HAL/driver logic	❌ Not recommended
Same-process HIDL Java call	❌ Not supported (only cross-process)

🧑‍💻 Example: IFoo Interface in Java

Suppose you have this HIDL interface:

```hidl
package android.hardware.foo@1.0;

interface IFoo {
    doSomething();
};
```

💡 HIDL Tool Command:

This command generates the Java interface:

```bash
hidl-gen -o /tmp -Ljava \
  -randroid.hardware:hardware/interfaces \
  -randroid.hidl:system/libhidl/transport \
  android.hardware.foo@1.0
```

It creates:
`/tmp/android/hardware/foo/1.0/IFoo.java`

🧩 How to Use IFoo as a Java Client

Add Library to Your Build:

Android.bp:

```bp
static_libs: ["android.hardware.foo-V1.0-java"],
```

Android.mk:

```mk
LOCAL_STATIC_JAVA_LIBRARIES += android.hardware.foo-V1.0-java
```

In Java code:

```java
import android.hardware.foo.V1_0.IFoo;

...

// Get default instance (registered as "default")
IFoo fooService = IFoo.getService(true); // retry=true to wait if it's not yet started

// OR get instance named "second_impl"
IFoo otherFoo = IFoo.getService("second_impl", true);

// Now call methods
fooService.doSomething();
```

⚠️ **`getService(true)`** waits until the service is available.
⚠️ **`getService()`** with **no arguments** might return null if service is not ready.

🔁 How to Use Callbacks from HAL in Java

Suppose you have a callback interface:

```hidl
interface IFooCallback {
    onEvent(uint32_t eventId);
};
```

Java Side (Client Receives Callback):

```java
import android.hardware.foo.V1_0.IFooCallback;

class FooCallback extends IFooCallback.Stub {
    @Override
    public void onEvent(int eventId) {
        Log.d("FOO", "Event received: " + eventId);
    }
}
```

Now set it to the HAL:

```java
FooCallback mFooCallback = new FooCallback(); // keep reference!
IFoo fooService = IFoo.getService(true);
fooService.setFooCallback(mFooCallback);  // Register your callback
```

⚠️ Keep **`mFooCallback`** as a member variable. If it's a local variable, **Java GC may remove it**, and callback won't work!

🔁 Interface Extensions in Java

Suppose you extend IFoo:

```hidl
interface IFoo { ... }
interface IBetterFoo extends IFoo { ... }
```

You can cast from **`IFoo`** to **`IBetterFoo`** if supported:

```java
IFoo base = IFoo.getService(true);
IBetterFoo better = IBetterFoo.castFrom(base);

if (better != null) {
    better.doAdvancedStuff();
} else {
    // Device doesn't support IBetterFoo
}
```

🧱 About Shallow Java Libraries

Starting in **Android 10**:

- HIDL Java libs split into:
  - **Full** (contains all dependencies)
  - **Shallow** (just this package)

Use shallow if you already include dependencies elsewhere:

```bp
static_libs: ["android.hardware.foo-V1.0-java-shallow"]
```

❗ Recommendations

- ✅ You can write **Java clients** for HIDL interfaces
- ✅ You can implement **callback receivers** in Java
- ⚠️ Don't implement HAL drivers in Java
- ⚠️ HIDL Java services must run in **separate processes**
- ❌ No same-process HIDL Java calls

Would you like a full **example Java project** using a HIDL interface and callback?

56. ↪
🔁 How to Use Callbacks from HAL in Java -------------------------------------- Suppose you have a callback interface: h CopyEdit `interface IFooCallback { onEvent(uint32_t eventId); };`
how hidl know it's callback

Great question!

HIDL **knows it's a callback** simply because **you (the developer) pass an interface object (e.g., `IFooCallback`) as a parameter to another interface's method** — this usage pattern **implicitly tells HIDL**: "This is a callback".

🔁 What defines a callback in HIDL?

In HIDL, **any interface passed as a parameter** to another interface method is **treated as a callback**.

📘 Example:

```hidl
// File: IFooCallback.hal
package android.hardware.foo@1.0;

interface IFooCallback {
    onEvent(uint32_t eventId);
};
```

```hidl
// File: IFoo.hal
package android.hardware.foo@1.0;

import IFooCallback;

interface IFoo {
    setFooCallback(IFooCallback callback);
};
```

Here, **`setFooCallback(IFooCallback callback);`** tells HIDL:

> The **`callback`** parameter is a reference to another HIDL interface — so this is a **callback mechanism**.

✅ Why is this automatic?

Because HIDL is **type-aware and transport-aware**, and:

- All interfaces in HIDL derive from **`android.hidl.base@1.0::IBase`**
- When you pass an **`interface`** type (like **`IFooCallback`**) to a method, HIDL:
  - Recognizes it as a **remote object** (like Binder)
  - Marshals it across processes
  - Allows the callee (server/HAL) to **store it and call back** into the caller (client/framework)

🧠 How does the HAL know how to call back?

Once it receives the **`IFooCallback`** object in **`setFooCallback()`**, the HAL stores it like this (in C++):

```cpp
sp<IFooCallback> mCallback;

Return<void> setFooCallback(const sp<IFooCallback>& callback) override {
    mCallback = callback;
    return Void();
}
```

Then later, it can do:

```cpp
mCallback->onEvent(123); // Async call back into client
```

🔁 Summary

Feature	Mechanism
How callback is declared	Pass interface as parameter to another interface
How HIDL knows it's a callback	It sees you're passing a HIDL interface as an argument
Runtime behavior	Binder transports the callback stub to the other side
Who calls whom	Server/HAL **calls back** into client using stored **`sp<IFooCallback>`**

Want a full Java + C++ example with real callbacks working end-to-end?

57. ↪
🧱 About Shallow Java Libraries ------------------------------ Starting in **Android 10**: * HIDL Java libs split into: * **Full** (contains all dependencies) * **Shallow** (just this package)
what is this

That refers to **"shallow Java libraries"** for HIDL interfaces in **Android 10 and above**.

🔍 What is a shallow Java library?

A **shallow Java library** for a HIDL interface is a **partial version** of the full HIDL Java library. It:

- **Contains only the classes/interfaces for a single HIDL package**
- **Does NOT include** its **dependencies**, like base interfaces (`android.hidl.base@1.0`) or types from other packages

---

🧱 Full vs. Shallow Java libraries

Type	Includes...	Use case
Full	Interface + all transitive dependencies (base interfaces, etc.)	Apps that need everything bundled
Shallow	Only that package's code	Framework modules that already depend on base HIDL libs, or save space

---

📦 Why Android introduced this?

In **Android 10**, the **bootclasspath** (core Java classpath for system apps) was tightened. Large monolithic HIDL libraries were wasting space and introducing unnecessary dependencies. So:

- Modules on the bootclasspath (like `framework.jar`) must avoid pulling in **duplicate or unused classes**
- Hence, they include only the **shallow version** and let the existing dependencies provide the rest (like `android.hidl.base@1.0-java`)

✅ When to use what?

Scenario	Library to use
System app or Java HAL using HIDL	**Full** Java lib (`-V1.0-java`)
Framework code (e.g. services.jar)	**Shallow** version (`-V1.0-java-shallow`)
Custom module with all deps already available	Shallow (to save space)

📌 Example

For a HIDL package like:

```css
android.hardware.foo@1.0
```

You get:

- `android.hardware.foo-V1.0-java` → **full** library
- `android.hardware.foo-V1.0-java-shallow` → **shallow** library

In `Android.bp`, if you are in `frameworks/base/`, you might write:

```bp
static_libs: [
    "android.hardware.foo-V1.0-java-shallow",
]
```

Let me know if you want a visual diagram or a real example using shallow vs. full!

58. Data types

bookmark_border
Given a HIDL interface file, the Java HIDL backend generates Java interfaces, Stub, and Proxy code. It supports all scalar HIDL types ([u]int{8,16,32,64}_t, float, double, and enums), as well as strings, interfaces, safe_union types, struct types, and arrays and vectors of supported HIDL types. The Java HIDL backend does NOT support union types or fmq types. Android 11 adds support for the memory and handle types.

As Java runtime doesn't support the concept of unsigned integers natively, all unsigned types (and enums based on them) are silently treated as their signed equivalents, i.e. uint32_t becomes an int in the Java interface. No value conversion is performed; the implementor on the Java side must use the signed values as if they were unsigned.

Enums
Enums don't generate Java enum classes but are instead translated into inner classes containing a static constant definition for each enum case. If the enum class derives from some other enum class, it inherits that class's storage type. Enumerations based on an unsigned integer type are rewritten into their signed equivalent. Since the underlying type is a primitive, the default value for enum fields/variables is zero even when there is no zero enumerator.

For example, a SomeBaseEnum with a type of uint8_t:

```
enum SomeBaseEnum : uint8_t { foo = 3 };
enum SomeEnum : SomeBaseEnum {
    quux = 33,
    goober = 127
};
```
... becomes:

```
public final class SomeBaseEnum { public static final byte foo = 3; }
public final class SomeEnum {
    public static final byte foo = 3;
    public static final byte quux = 33;
    public static final byte goober = 127;
}
```
And:

```
enum SomeEnum : uint8_t {
    FIRST_CASE = 10,
    SECOND_CASE = 192
};
```
... is rewritten as:

```
public final class SomeEnum {
    static public final byte FIRST_CASE  = 10;  // no change
    static public final byte SECOND_CASE = -64;
}
```
Strings
String in Java is utf-8 or utf-16 but is converted to utf-8 as the common HIDL type when transported. Additionally, String must not be null when passed into HIDL.

Handle and memory
Android 11 introduces Java support for the handle and memory types. They are translated into android.os.NativeHandle and android.os.HidlMemory, respectively. A null handle is considered valid, while a null memory isn't.

In the generated server code, received memory and handle arguments are only valid within the scope of the method invocation. If the server implementation wants to extend their lifetime, they must be duplicated using their resepective dup() methods. The returned instance can be used beyond the method invocation and should be properly closed when done with.

In the generated client code, handles and memory instances that are sent as input arguments of the called method don't need to be duplicated nor kept valid after the method returns. However, handles and memory instances that are received as output arguments are automatically duplicated by the autogenerated code and must be properly closed when done with. This is true whether those return arguments appear as return values of the method (in the single return value case) or using the synchronous callback style (used in the multiple return value case).

For more information about duplication and closing, see the documentation of the Java classes.

Arrays and vectors
Arrays are translated into Java arrays and vectors are translated into ArrayList<T> where T is the appropriate object type, possibly wrapping scalar types such as vec<int32_t> => ArrayList<Integer>). For example:

```
takeAnArray(int32_t[3] array);
returnAVector() generates (vec<int32_t> result);
... becomes:
```

```
void takeAnArray(int[] array);
ArrayList<Integer> returnAVector();
```

Structures
Structures are translated into Java classes with a similar layout. For example:

```
struct Bar {
 vec<bool> someBools;
};
struct Foo {
 int32_t a;
 int8_t b;
 float[10] c;
 Bar d;
};
... becomes:
```

```
class Bar {
 public final ArrayList<Boolean> someBools = new ArrayList();
};
class Foo {
 public int a;
 public byte b;
 public final float[] c = new float[10];
 public final Bar d = new Bar();
}
```

Declared types
Each top-level type declared in types.hal gets its own .java output file (as required by Java). For example, the following types.hal file results in two extra files being created (Foo.java and Bar.java):

```
struct Foo {
 ...
};

struct Bar {
 ...

 struct Baz {
 };

 ...
};
```

The definition of Baz lives in a static inner class of Bar (in Bar.java).

Here's a **simple explanation** of the HIDL **Java data type mapping**, based on what you posted:

🔁 Overview

The HIDL Java backend **generates Java interface code** from `.hal` files. It supports almost all common types, with some differences due to **Java's type system** (especially around unsigned types and memory handling).

🧠 Java Support Summary

HIDL Type	Java Equivalent	Notes
`uint32_t`, `int32_t`	`int`	Java has no unsigned types. You must handle wrapping manually.
`float`, `double`	`float`, `double`	Same
`string`	`String`	Must **not be null**. Uses UTF-8
`vec<T>`	`ArrayList<T>`	e.g. `vec<int32_t>` → `ArrayList<Integer>`
`T[N]` (array)	`T[]`	Fixed-length array
`struct`	Java `class`	Maps to public fields
`enum`	Static final fields (not Java `enum`)	e.g. `public final static byte FOO = 1`
`handle`	`android.os.NativeHandle`	Only available from Android 11
`memory`	`android.os.HidlMemory`	Only available from Android 11
`interface`	Java interface + Stub/Proxy	Like AIDL-style interface
`safe_union`	Java class with one active field	Limited support
`union`, `fmq`	❌ Not supported in Java	Use C++ HIDL instead

🔢 Unsigned Integer Caveat

Java doesn't support unsigned types like `uint8_t`, `uint32_t`. So:

```hidl
enum MyEnum : uint8_t {
  CASE_1 = 10,
  CASE_2 = 192
};
```

Becomes:

```java
public final class MyEnum {
    public static final byte CASE_1 = 10;      // OK
    public static final byte CASE_2 = -64;     // Wrapped from 192 to signed byte
}
```

📌 You have to **treat signed values as if they were unsigned** when needed.

🏗️ Struct to Java Class Example

HIDL:

```hidl
struct Foo {
    int32_t a;
    int8_t b;
    float[10] c;
    vec<bool> someBools;
};
```

Java:

```java
public class Foo {
    public int a;
    public byte b;
    public final float[] c = new float[10];
    public final ArrayList<Boolean> someBools = new ArrayList<>();
}
```

🧱 Nested Types

If your `.hal` defines inner structs:

```hidl
struct Bar {
  struct Baz {
    int32_t x;
  };
};
```

Java will generate:

```java
public class Bar {
    public static class Baz {
        public int x;
    }
}
```

📁 File Structure

If your HIDL file is:

```cpp
types.hal:
  struct Foo
  struct Bar
```

Then **`hidl-gen`** creates:

```
Foo.java
Bar.java
```

If **`Bar`** has a nested **`Baz`**, it becomes:

```java
public class Bar {
    public static class Baz { ... }
}
```

🧪 Arrays and Vectors

```hidl
takeAnArray(int32_t[3] arr);
returnAVector() generates (vec<int32_t>);
```

→ Becomes:

```java
void takeAnArray(int[] arr);
ArrayList<Integer> returnAVector();
```

🧷 Handle and Memory (Android 11+)

HIDL:

```hidl
someMethod(handle h, memory m);
```

Java:

```java
void someMethod(NativeHandle h, HidlMemory m);
```

> ✅ You must call `.dup()` if you want to use the handle/memory outside the method — to **own** the resource and prevent GC issues.

❓ Still unclear?

I can:

- Map a full `.hal` to its Java output
- Show an example project structure
- Visualize type conversion

Let me know how deep you want to go!

59. ↪
enum MyEnum : uint8_t { CASE_1 = 10, CASE_2 = 192 };
explain this syntax

Let's break down this HIDL syntax:

```hidl
enum MyEnum : uint8_t {
    CASE_1 = 10,
    CASE_2 = 192
};
```

✅ Meaning of Each Part

`enum MyEnum`

Defines a **named enumeration** called **`MyEnum`**. An **`enum`** is a set of **named constant values**, useful for clarity and type safety.

`: uint8_t`

This specifies the **underlying storage type** for the enum — here it's an **8-bit unsigned integer**.

So **`MyEnum`** will be backed by a **`uint8_t`** (values between **`0`** and **`255`**).

In C++/HIDL:

```c++
MyEnum val = MyEnum::CASE_1;
uint8_t raw = static_cast<uint8_t>(val);  // raw = 10
```

`{ CASE_1 = 10, CASE_2 = 192 }`

These are the **enumerators** (named values). You assign specific values:

- **`CASE_1`** has value 10
- **`CASE_2`** has value 192

📦 Purpose in HIDL

You use enums in HIDL to define **limited, meaningful sets of values** in your HAL interface. These get auto-converted to constants in generated C++ and Java code.

👇 Java Output (converted version)

Since Java doesn't support **`uint8_t`**, this enum will be converted like this:

```java
public final class MyEnum {
    public static final byte CASE_1 = 10;
    public static final byte CASE_2 = -64; // ⚠ because 192 overflows signed byte
}
```

> ✅ Java **`byte`** = signed 8-bit (range -128 to 127). So **`192`** becomes **`-64`** in Java. You must handle such values carefully.

📘 Summary

Part	Meaning
`enum MyEnum`	Defines an enum named **`MyEnum`**
`: uint8_t`	Backed by unsigned 8-bit integer (0–255)
`CASE_1 = 10`	Named constant with value **`10`**
`CASE_2 = 192`	Named constant with value **`192`**
Java equivalent	Final constants of type **`byte`**

Let me know if you want a visual or usage example in a HIDL method!

60. Interface methods and errors

bookmark_border
This section details interface methods and errors.

Void methods
Methods that don't return results are translated into Java methods that return void. For example, the HIDL declaration:

```
doThisWith(float param);
```
... becomes:

```
void doThisWith(float param);
```
Single-result methods
Methods that return a single result are translated into their Java equivalents also returning a single result. For example, the following:

```
doQuiteABit(int32_t a, int64_t b,
            float c, double d) generates (double something);
```
... becomes:

```
double doQuiteABit(int a, long b, float c, double d);
```
Multiple-result methods
For each method that returns more than one result, a callback class is generated that supplies all the results in its onValues method. That callback acts as an additional parameter to the method. For example, the following:

```
oneProducesTwoThings(SomeEnum x) generates (double a, double b);
```
... becomes:

```
public interface oneProducesTwoThingsCallback {
    public void onValues(double a, double b);
}
void oneProducesTwoThings(byte x, oneProducesTwoThingsCallback cb);
```
A caller of oneProducesTwoThings() would typically use an anonymous inner class or lambda to implement the callback locally:

```
someInstanceOfFoo.oneProducesTwoThings(
         5 /* x */,
         new IFoo.oneProducesTwoThingsCallback() {
          @Override
          void onValues(double a, double b) {
             // do something interesting with a and b.
             ...
          }});
```
or:

```
someInstanceOfFoo.oneProducesTwoThings(5 /* x */,
    (a, b) -> a > 3.0 ? f(a, b) : g(a, b)));
```
You can also define a class to use as a callback ...

```
class MyCallback implements oneProducesTwoThingsCallback {
  public void onValues(double a, double b) {
    // do something interesting with a and b.
  }
}
```
... and pass an instance of MyCallback as the third parameter to oneProducesTwoThings().

Transport errors and death recipients
Because service implementations can run in a different process, in some cases the client can stay alive even when the process implementing an interface dies. Calls on an interface object hosted in a dead process fail with a transport error (a runtime exception thrown by the called method). Recovery from such a failure is possible by requesting a new instance of the service by calling I<InterfaceName>.getService(). However, this method works only if the process that crashed has restarted and re-registered its services with the servicemanager (which is generally true for HAL implementations).

Clients of an interface can also register a death recipient to get a notification when a service dies. Transport errors can still occur if a call is made just as the server dies. To register for such notifications on a retrieved IFoo interface, a client can do the following:

foo.linkToDeath(recipient, 1481 /* cookie */);
The recipient parameter must be an implementation of the interface HwBinder.DeathRecipient provided by HIDL. The interface contains a single method serviceDied() that is called when the process hosting the interface dies.

```
final class DeathRecipient implements HwBinder.DeathRecipient {
    @Override
    public void serviceDied(long cookie) {
        // Deal with service going away
    }
}
```
The cookie parameter contains the cookie that was passed with the call to linkToDeath(). It's also possible to unregister a death recipient after registering it using:

foo.unlinkToDeath(recipient);

Let's break this down and explain each concept and syntax from your last message in **simple, clear parts**:

🔹 1. Void Methods

HIDL:

```hidl
doThisWith(float param);
```

Means: A method that **takes a float** and **returns nothing**.

Java equivalent:

```java
void doThisWith(float param);
```

- Just a simple function.
- No callback, no return value.

🔹 2. Single-result Methods

HIDL:

```hidl
doQuiteABit(int32_t a, int64_t b, float c, double d) generates (double something);
```

Means: Function takes four arguments and **returns one value (`double`)**.

Java equivalent:

```java
double doQuiteABit(int a, long b, float c, double d);
```

- HIDL auto-generates a **normal Java method** that returns a single value.
- No callback needed.
- Directly usable like any regular Java method.

🔹 3. Multiple-result Methods

HIDL:

```hidl
oneProducesTwoThings(SomeEnum x) generates (double a, double b);
```

Means: Takes one input (`SomeEnum`), and **returns two values**.

Java equivalent:

You **can't return multiple values directly in Java**, so HIDL Java backend creates a **callback interface**:

```java
public interface oneProducesTwoThingsCallback {
    void onValues(double a, double b);
}

void oneProducesTwoThings(byte x, oneProducesTwoThingsCallback cb);
```

✅ Usage examples:

Option 1: Anonymous class

```java
foo.oneProducesTwoThings(5, new IFoo.oneProducesTwoThingsCallback() {
    @Override
    public void onValues(double a, double b) {
        // use a and b here
    }
});
```

Option 2: Lambda (if callback is a functional interface)

```java
foo.oneProducesTwoThings(5, (a, b) -> {
    // use a and b here
});
```

Option 3: Reusable class

```java
class MyCallback implements IFoo.oneProducesTwoThingsCallback {
    public void onValues(double a, double b) {
        // handle values
    }
}

foo.oneProducesTwoThings(5, new MyCallback());
```

🔹 4. Transport Errors & Death Recipients

Problem:

The HAL server process might **crash or die**, and your client could still be alive. Then calling a method causes a **runtime exception**.

Solution:

Register a **death recipient** to get notified when the HAL dies.

Example:

```java
final class DeathRecipient implements HwBinder.DeathRecipient {
    @Override
    public void serviceDied(long cookie) {
        // This is called when the HAL process dies
    }
}
```

Register the death recipient:

```java
IFoo foo = IFoo.getService();
foo.linkToDeath(new DeathRecipient(), 1481); // 1481 is a user-defined cookie
```

Optional: Unregister

```java
foo.unlinkToDeath(recipient);
```

🔚 Summary Table

HIDL Definition	Java Signature / Behavior
`doThisWith(float)`	`void doThisWith(float)`
`generates (double something)`	`double do...()`
`generates (double a, double b)`	Callback interface with `onValues(double, double)`
`linkToDeath()`	Detect if HAL service dies

Let me know if you want me to generate a working client-server Java sample using a HIDL definition!

61. Export constants

bookmark_border
In cases where an interface isn't Java-compatible (because it uses unions for example) it might still be desirable to export the constants (enum values) to the Java world. This scenario is supported by hidl-gen -Ljava-constants … which extracts annotated enum declarations from the interface file(s) in a package and produces a java library named [PACKAGE-NAME]-V[PACKAGE-VERSION]-java-constants. Annotate each enum declaration to be exported as follows:

```
@export
enum Foo : int32_t {
  SOME_VALUE,
  SOME_OTHER_VALUE,
};
```

If necessary, the name under which this type is exported to the Java world can be different from that chosen in the interface declaration by adding the annotation-parameter name:

```
@export(name="JavaFoo")
enum Foo : int32_t {
  SOME_VALUE,
  SOME_OTHER_VALUE,
};
```

If Java conventions or personal preference ask for a common prefix to be added to the enum type's values, use the annotation-parameter value_prefix:

```
// File "types.hal".

package android.hardware.bar@1.0;

@export(name="JavaFoo", value_prefix="JAVA_")
enum Foo : int32_t {
  SOME_VALUE,
  SOME_OTHER_VALUE,
};
```

The resulting Java class appears as follows:

```
package android.hardware.bar.V1_0;

public class Constants {
  public final class JavaFoo {
    public static final int JAVA_SOME_VALUE = 0;
    public static final int JAVA_SOME_OTHER_VALUE = 1;
  };
};
```

Finally, Java type declaration for enum types declared in types.hal are grouped inside a class Constants in the given package. Enum types declared as children of an interface are grouped under that interface's Java class declaration.

This section is about **exporting HIDL enums to Java** as **constants**, even when the HIDL interface cannot be fully used in Java (e.g., it uses `union` or unsupported features).

Let me break this down in a clear and simple way:

🟩 Why is this needed?

Some HIDL types like `union` or `fmq` are **not supported in Java**.

But even if you **can't use the whole interface in Java**, you may still want to use **constants** like enum values in your Java code.

✅ Example use case:
Use HIDL-defined flags or options in Java app logic or system service.

🛠️ Solution: Export Only the Constants

You can export enum values as constants to Java using the HIDL annotation:

```hidl
@export
enum Foo : int32_t {
  SOME_VALUE,
  SOME_OTHER_VALUE,
};
```

Then run the generator:

```sh
hidl-gen -o output/dir -Ljava-constants ...
```

This will create a Java class library:

```css
[PACKAGE-NAME]-V[VERSION]-java-constants
```

🔄 Customizing the Java Output

1. Rename the Enum Type in Java

```hidl
@export(name="JavaFoo")
enum Foo : int32_t {
  SOME_VALUE,
  SOME_OTHER_VALUE,
};
```

➡️ In Java this becomes:

```java
public final class JavaFoo {
  public static final int SOME_VALUE = 0;
  public static final int SOME_OTHER_VALUE = 1;
}
```

---

2. Add a Prefix to Enum Values in Java

```hidl
@export(name="JavaFoo", value_prefix="JAVA_")
enum Foo : int32_t {
  SOME_VALUE,
  SOME_OTHER_VALUE,
};
```

➡️ In Java this becomes:

```java
public final class JavaFoo {
  public static final int JAVA_SOME_VALUE = 0;
  public static final int JAVA_SOME_OTHER_VALUE = 1;
}
```

✅ Helpful for Java naming convention or avoiding naming conflicts.

📦 Final Java Structure

Java constants are **grouped inside a `Constants` class** based on their HIDL package.

HIDL:

```hidl
package android.hardware.bar@1.0;
```

Java result:

```java
package android.hardware.bar.V1_0;

public class Constants {
  public final class JavaFoo {
    public static final int JAVA_SOME_VALUE = 0;
    public static final int JAVA_SOME_OTHER_VALUE = 1;
  }
}
```

You access them in Java like:

```java
int val = android.hardware.bar.V1_0.Constants.JavaFoo.JAVA_SOME_VALUE;
```

✅ Summary

Feature	HIDL Syntax	Java Output
Export enum	`@export`	Java constants
Rename enum	`@export(name="JavaFoo")`	Class name is `JavaFoo`
Add prefix	`@export(value_prefix="JAVA_")`	Values become `JAVA_...`
Java wrapper	–	Constants are grouped inside `Constants` class per package

---

Let me know if you want a working example with `hidl-gen` command, `Android.bp`, and generated Java file!

62. confusing

No worries — let's break it down **step by step in the simplest terms**:

🟦 Problem

You wrote a HIDL file like this:

```hidl
enum Foo : int32_t {
  SOME_VALUE,
  SOME_OTHER_VALUE,
};
```

You want to **use `SOME_VALUE` and `SOME_OTHER_VALUE` in Java**, **without using the full HIDL interface** (because maybe Java doesn't support `union`, `fmq`, etc.).

🟩 Solution

Use `@export` annotation so HIDL can **generate only the constants** in Java.

✅ Step-by-step:

🔹 1. Mark enum with `@export`:

```hidl
@export
enum Foo : int32_t {
  SOME_VALUE,
  SOME_OTHER_VALUE,
};
```

This tells the HIDL tool:

> "Please export these values as Java constants."

🔹 2. Optional: Rename it for Java

```hidl
@export(name="JavaFoo")
enum Foo : int32_t {
  SOME_VALUE,
  SOME_OTHER_VALUE,
};
```

➡️ This will generate a **Java class** called **`JavaFoo`**.

🔹 3. Optional: Add a prefix for values

```hidl
@export(name="JavaFoo", value_prefix="JAVA_")
enum Foo : int32_t {
  SOME_VALUE,
  SOME_OTHER_VALUE,
};
```

➡️ This will create Java constants like:

```java
public final class JavaFoo {
  public static final int JAVA_SOME_VALUE = 0;
  public static final int JAVA_SOME_OTHER_VALUE = 1;
}
```

🔹 4. Run `hidl-gen` to generate Java code

```bash
hidl-gen -o out/dir -Ljava-constants \
  -randroid.hardware:hardware/interfaces \
  android.hardware.foo@1.0
```

🔹 5. Use in Java

Assuming your HIDL package is **`android.hardware.foo@1.0`**, you can use it in Java like:

```java
import android.hardware.foo.V1_0.Constants.JavaFoo;

int x = JavaFoo.JAVA_SOME_VALUE;
```

🟧 Visual Summary

HIDL File	Generated Java Code
`@export`	**`Constants.JavaFoo.SOME_VALUE`**
`@export(name="JavaFoo")`	class name is **`JavaFoo`**
`value_prefix="JAVA_"`	values are **`JAVA_SOME_VALUE`**, etc.

💡 Tip

You can **use this even if your interface is too complex for Java** — just to get the constants.

Do you want me to show a full **`types.hal`**, **`Android.bp`**, and Java usage together as a working example?

63. HIDL Framework backward compatibility verification

bookmark_border
HIDL HALs guarantee the Android core system (aka system.img or the framework) is backward compatible. While Vendor Test Suite (VTS) tests ensure that HALs work as expected (e.g. 1.1 HAL tests are run on all 1.2 implementations), framework testing is needed to ensure that when a supported HAL (1.0, 1.1, or 1.2) is provided, the framework works properly with that HAL.

For details on HAL interface definition language (HIDL), refer to HIDL, HIDL versioning, and HIDL HAL Deprecation.

About HAL upgrades
There are two types of HAL upgrades: major and minor. Most systems include only one HAL implementation, but multiple implementations are supported. For example:

android.hardware.teleport@1.0 # initial interface
android.hardware.teleport@1.1 # minor version upgrade
android.hardware.teleport@1.2 # another minor version upgrade
...
android.hardware.teleport@2.0 # major version upgrade
...
The system partition typically includes a framework daemon (such as teleportd) that manages communication with a specific group of HAL implementations. Alternatively, systems might instead include a system library (such as android.hardware.configstore-utils) that implements convenient client behavior. In the example above, teleportd must work no matter what version of the HAL is installed on the device.

Let's simplify and clearly explain this topic of **HIDL framework backward compatibility verification**:

🟦 What is the issue?

Android uses **HIDL** to define how the **framework (system.img)** talks to **vendor-specific HALs**.

As Android evolves, new versions of HALs may be introduced:

- `@1.0` → initial version
- `@1.1`, `@1.2` → minor upgrades
- `@2.0` → major upgrade

The key question is:

> **How do we ensure that the Android system framework still works with older HAL versions (like 1.0, 1.1)?**

✅ HIDL's Promise

> **The framework must be backward compatible with older HAL versions.**

That means:

- A framework written for HAL version 1.2 must still work correctly if the device only has HAL 1.0 or 1.1.

🔄 HAL Upgrade Types

Type	What changes?	Compatibility requirement
Minor	New methods added, old methods untouched	Must work with older HALs
Major	Big changes: method removed, types changed	Not backward-compatible

Example progression:

```less
@1.0 → basic features
@1.1 → added new method, kept 1.0 methods
```

```
@1.2 → added another method, still compatible
@2.0 → removed/renamed stuff — breaking change!
```

🔧 How is compatibility verified?

🔸 1. Vendor Test Suite (VTS)

- Verifies that HAL **implementation** is valid.
- Runs **old version tests (1.0, 1.1)** even if HAL is 1.2.
- Ensures **HALs are backward-compatible**.

But **VTS alone is not enough**.

🔸 2. Framework Compatibility Testing

This checks the **framework side**:

> "If you provide HAL 1.0, does the system server (e.g., `teleportd`) still function correctly?"

- System components (like `teleportd`, which talks to the teleport HAL) must **gracefully degrade** if a newer feature is missing.
- The framework must **adapt dynamically** based on the available HAL version.

📦 Example

Imagine a HAL called `android.hardware.teleport@1.x` used for sci-fi teleportation.

You might have:

```sql
android.hardware.teleport@1.0
  - method: startTeleport()

android.hardware.teleport@1.1
  - method: startTeleport()
  - method: cancelTeleport()

android.hardware.teleport@1.2
  - method: startTeleport()
  - method: cancelTeleport()
  - method: setDestination()

android.hardware.teleport@2.0
  - different structure; incompatible
```

A framework process (`teleportd`) must:

- Check version of HAL available
- Only call `cancelTeleport()` if HAL ≥ 1.1
- Only call `setDestination()` if HAL ≥ 1.2

🔍 Summary

Aspect	Description
Goal	Ensure framework can still use older HALs correctly
Who tests HAL?	VTS (Vendor Test Suite)
Who tests framework?	Compatibility test by system developers
Why needed?	Prevent system breakage on devices with older HALs
Key rule	Never assume the newest HAL is present

If you'd like, I can show you code examples of how a framework checks for HAL version before calling newer APIs. Want that?

64. Dynamically available HALs

bookmark_border
Android 9 supports the dynamic shutdown of Android hardware subsystems when they are not in use or not needed. For example, when a user is not using Wi-Fi, the Wi-Fi subsystems should not be taking up memory, power, or other system resources. In earlier versions of Android, HALs/drivers were kept open on Android devices for the entire duration an Android phone was booted.
Implementing dynamic shutdown involves wiring up data flows and executing dynamic processes as detailed in the following sections.

Changes to HAL definitions
Dynamic shutdown requires information on which processes serve what HAL interfaces (this information may also be useful later in other contexts) as well as not starting processes on boot and not restarting them (until requested again) when they exit.

```
# some init.rc script associated with the HAL
service vendor.some-service-name /vendor/bin/hw/some-binary-service
    # init language extension, provides information of what service is served
    # if multiple interfaces are served, they can be specified one on each line
    interface android.hardware.light@2.0::ILight default
    # restarted if hwservicemanager dies
    # would also cause the hal to start early during boot if disabled wasn't set
    class hal
    # will not be restarted if it exits until it is requested to be restarted
    oneshot
    # will only be started when requested
    disabled
    # ... other properties
```

Changes to init and hwservicemanager
Dynamic shutdown also requires the hwservicemanager to tell init to start requested services. In Android 9, init includes three additional control messages (e.g. ctl.start): ctl.interface_start, ctl.interface_stop, and ctl.interface_restart. These messages can be used to signal init to bring up and down specific hardware interfaces. When a service is requested and isn't registered, hwservicemanager requests that the service be started. However, dynamic HALs don't require using any of these.

Determine HAL exit
In Android 9, HAL exit has to be manually determined. For Android 10 and higher, it can also be determined with automatic lifecycles.

Dynamic shutdown requires multiple policies for deciding when to start a HAL and when to shutdown a HAL. If a HAL decides to exit for any reason, it will automatically be restarted when it is needed again using the information provided in the HAL definition and the infrastructure provided by changes to init and hwservicemanager. This could involve a couple of different strategies, including:

A HAL could choose to call exit on itself if someone calls a close or similar API on it. This behavior must be specified in the corresponding HAL interface.
HALs can shut down when their task is completed (documented in the HAL file).
Automatic lifecycles
Android 10 adds more support to the kernel and hwservicemanager, which allows HALs to shut down automatically whenever they have no clients. To use this feature, do all of the steps in Changes to HAL definitions as well as:

Register the service in C++ with LazyServiceRegistrar instead of the member function, registerAsService, for example:

```
// only one instance of LazyServiceRegistrar per process
LazyServiceRegistrar registrar;
registrar.registerAsService(myHidlService /* , "default" */);
```

Verify that the HAL client keeps a reference to the top-level HAL (the interface registered with hwservicemanager) only when it's in use. To avoid delays if this reference is dropped on a hwbinder thread that continues to execute, the client should also call IPCThreadState::self()->flushCommands() after dropping the reference to ensure that the binder driver is notified of the associated reference count changes.
Was this helpful?

Yes — here's a **simple explanation** of **Dynamically Available HALs** (Dynamic HAL shutdown) and what's going on:

🚦 What's the Problem?

In **old Android versions**, once a **HAL service** (like Wi-Fi, Bluetooth, Light, etc.) started during boot, it **never stopped** — even if unused.
This **wastes memory, battery, and CPU cycles.**

✅ What's the Solution?

Since **Android 9**, Android supports **Dynamic HALs** — HALs that can:

- ❌ **Not start on boot**
- ✅ **Start on demand**
- ✅ **Shut down automatically** when not in use

🔧 How to Enable This? (Step-by-Step)

🛠️ 1. init.rc configuration (service definition)

In your **`init.rc`**, you need:

```rc
service vendor.some-service-name /vendor/bin/hw/some-binary-service
    interface android.hardware.light@2.0::ILight default
    class hal
    oneshot
    disabled
```

🔍 Explanation:

- **`interface`**: declares the HAL interface being served
- **`oneshot`**: means **don't restart** after it exits (unless requested)
- **`disabled`**: means **don't start at boot**
- **`class hal`**: groups HAL services

🔧 2. hwservicemanager + init interaction

- When a client (like system server or app) calls **`ILight::getService("default")`**, but the HAL isn't running:
  - **`hwservicemanager`** sends **`ctl.interface_start`** to **init**
  - **`init`** starts the right HAL service
  - HAL gets registered, client proceeds

✔️ No manual start/stop needed!

⚠️ 3. How HAL knows when to exit?

Two options:

Option 1: Manual exit

You write code to call **`exit(0)`** when:

- Some **`close()`** method is called, OR
- A task completes

Option 2: 🔁 Automatic Lifecycle (Android 10+)

- Use **`LazyServiceRegistrar`** in the HAL code:

```cpp
LazyServiceRegistrar registrar;
registrar.registerAsService(myService);
```

- HAL exits **automatically** when **no clients are connected**
- Android's binder system + `hwservicemanager` keeps track of clients
- When someone connects later, HAL is auto-started again

☝️ Important Tip for Clients

After releasing the HAL object, client must do:

```cpp
IPCThreadState::self()->flushCommands();
```

🔍 This tells binder: "I dropped the reference, so HAL can shut down now."

💡 Summary Table

Feature	Android 8	Android 9	Android 10+
Always-running HALs	✅	❌	❌
Start-on-demand	❌	✅	✅
Auto shutdown	❌	Manual only	✅ (Lazy HALs)
Uses LazyServiceRegistrar	❌	Optional	✅ Required for auto-shutdown
Reference-based lifecycle	❌	❌	✅

🧠 Key Benefit

This saves:

- 💾 Memory (no unused HALs running)
- 🔋 Battery (idle HALs not draining power)
- 🚀 Boot time (not starting all HALs at once)

Would you like a full code example for a **Lazy HAL with automatic shutdown** and `init.rc` + C++ HAL registration?

65. ↪
🔧 2. **hwservicemanager + init interaction** * When a client (like system server or app) calls `ILight::getService("default")`, but the HAL isn't running: * `hwservicemanager` sends `ctl.interface_start` to **init** * `init` starts the right HAL service * HAL gets registered, client proceeds ✔️ No manual start/stop needed!
explain in simple words

Sure! Here's a **very simple explanation** of what happens:

🤖 Problem:

Sometimes, a HAL (like for lights, camera, etc.) **isn't running** because it's turned off to save battery and memory.

But now an app or the system wants to **use it**.

✅ What happens step by step:

1. 🧑 App or system says:

   > "Hey, I want to use `ILight`"
   > (This is done by calling `ILight::getService("default")`)

2. 😕 But `ILight` HAL is not running right now.
3. 🧠 The **hwservicemanager** notices this and says:

   > "Okay, let me tell init to start it."

4. 📣 It sends a special message to **init**:

   ```
   ctl.interface_start
   ```

5. 🚀 `init` gets the message and **starts the HAL service**.
6. 🟢 The HAL is now running, and the app can use it!

🎉 Benefit:

- You **don't need to keep the HAL running all the time**
- It **starts automatically** when needed

Let me know if you want this shown with a diagram or a real example like `ILight` or `ICamera`.

66. A HAL defines a standard interface for hardware vendors to implement, which enables Android to be agnostic about lower-level driver implementations. Using a HAL allows you to implement functionality without affecting or modifying the higher level system. This page describes the older architecture, which is no longer supported as of Android 8.0. For Android 8.0 and higher, please see the HAL Overview.

HAL components
Figure 1. HAL components

You must implement the corresponding HAL (and driver) for the specific hardware your product provides. HAL implementations are typically built into shared library modules (.so files), but as Android does not mandate a standard interaction between a HAL implementation and device drivers, you can do what is best for your situation. However, to enable the Android system to correctly interact with your hardware, you must abide by the contract defined in each hardware-specific HAL interface.

To guarantee that HALs have a predictable structure, each hardware-specific HAL interface has properties defined in hardware/libhardware/include/hardware/hardware.h. This interface allows the Android system to load correct versions of your HAL modules in a consistent way. A HAL interface consists of two components: modules and devices.

HAL modules
A module represents your packaged HAL implementation, which is stored as a shared library (.so file). The hardware/libhardware/include/hardware/hardware.h header file defines a struct (hw_module_t) that represents a module and contains metadata such as the version, name, and author of the module. Android uses this metadata to find and load the HAL module correctly.

In addition, the hw_module_t struct contains a pointer to another struct, hw_module_methods_t, that contains a pointer to an open function for the module. This open function is used to initiate communication with the hardware for which the HAL is serving as an abstraction. Each hardware-specific HAL usually extends the generic hw_module_t struct with additional information for that specific piece of hardware. For example, in the camera HAL, the camera_module_t struct contains a hw_module_t struct along with other camera-specific function pointers:

```
typedef struct camera_module {
    hw_module_t common;
    int (*get_number_of_cameras)(void);
    int (*get_camera_info)(int camera_id, struct camera_info *info);
} camera_module_t;
```

When you implement a HAL and create the module struct, you must name it HAL_MODULE_INFO_SYM. Example from the Nexus 9 audio HAL:

```
struct audio_module HAL_MODULE_INFO_SYM = {
    .common = {
        .tag = HARDWARE_MODULE_TAG,
        .module_api_version = AUDIO_MODULE_API_VERSION_0_1,
        .hal_api_version = HARDWARE_HAL_API_VERSION,
        .id = AUDIO_HARDWARE_MODULE_ID,
        .name = "NVIDIA Tegra Audio HAL",
        .author = "The Android Open Source Project",
        .methods = &hal_module_methods,
    },
};
```

HAL devices
A device abstracts the hardware of your product. For example, an audio module can contain a primary audio device, a USB audio device, or a Bluetooth A2DP audio device.

A device is represented by the hw_device_t struct. Similar to a module, each type of device defines a detailed version of the generic hw_device_t that contains function pointers for specific features of the hardware. For example, the audio_hw_device_t struct type contains function pointers to audio device operations:

```
struct audio_hw_device {
    struct hw_device_t common;

    /**
     * used by audio flinger to enumerate what devices are supported by
     * each audio_hw_device implementation.
     *
     * Return value is a bitmask of 1 or more values of audio_devices_t
```

```
  */
  uint32_t (*get_supported_devices)(const struct audio_hw_device *dev);
  ...
};
typedef struct audio_hw_device audio_hw_device_t;
```

In addition to these standard properties, each hardware-specific HAL interface can define more of its own features and requirements. For details, see the HAL reference documentation as well as the individual instructions for each HAL.

Build HAL modules
HAL implementations are built into modules (.so) files and are dynamically linked by Android when appropriate. You can build your modules by creating Android.mk files for each of your HAL implementations and pointing to your source files. In general, your shared libraries must be named in a specific format so they can be found and loaded properly. The naming scheme varies slightly from module to module, but follows the general pattern of: <module_type>.<device_name>.

Legacy HAL
The term Legacy HAL refers broadly to all pre-Android 8.0 HALs (deprecated in Android 8). The bulk of Android system interfaces (camera, audio, sensors, etc.) are defined under `hardware/libhardware/include/hardware` and have rough versioning and a roughly stable ABI. A few subsystems (including Wi-Fi, Radio Interface Layer, and Bluetooth) have other non-standardized interfaces in `libhardware_legacy` or interspersed throughout the codebase. Legacy HALs never provided hard stability guarantees.

Here's a **simple explanation** of the **legacy HAL architecture (before Android 8.0)**:

🧩 What is a HAL?

HAL = **Hardware Abstraction Layer**

- It **connects Android** with **device-specific hardware drivers** (e.g. camera, audio, GPS).
- It **hides driver differences** so Android can work on different hardware without changes.

🏗️ Two main parts of a HAL:

1. HAL Module

- A **shared library (.so)** file, like: `audio.primary.tegra.so`
- Identified using `hw_module_t`
- Contains:
  - Metadata (name, version, author)
  - Pointer to methods (`open()` function)
  - Optional device-specific methods (e.g., `get_number_of_cameras()`)

```c
typedef struct camera_module {
    hw_module_t common; // base metadata
    int (*get_number_of_cameras)(void);
    ...
} camera_module_t;
```

2. HAL Device

- Represents the **actual hardware component**, like the primary audio device
- Uses `hw_device_t` as base
- Extended structs like `audio_hw_device_t` add specific function pointers:

```c
struct audio_hw_device {
    struct hw_device_t common;
    uint32_t (*get_supported_devices)(const struct audio_hw_device *dev);
    ...
};
```

🛠️ Building a HAL

- HALs are compiled as `.so` files using `Android.mk` files.
- They follow a naming convention:

```php-template
<hal_type>.<device_name>.so
e.g., audio.primary.tegra.so
```

🕰️ Legacy HALs (before Android 8.0)

- Were **not versioned properly**
- No strong ABI (Application Binary Interface) guarantees
- HALs lived in:
  - `hardware/libhardware/include/hardware` (main ones)
  - `libhardware_legacy` (Wi-Fi, RIL, Bluetooth, etc.)

❌ Why this is deprecated in Android 8.0+

- Legacy HALs had no strict versioning or stability.
- Android 8 introduced **HIDL HALs**, which are:
  - Versioned (e.g., `@1.0`, `@2.1`)
  - Language-neutral (Java/C++)
  - Easier to test and maintain
  - Designed to work well with Treble and modular system updates

---

🔁 Summary

Term	Meaning
`hw_module_t`	Base struct for a HAL module (e.g., audio, camera)
`hw_device_t`	Base struct for a HAL device (e.g., actual microphone)
Module	`.so` file that Android loads dynamically
Device	The real hardware logic used inside the HAL
Legacy HAL	Pre-Android 8 HAL style; not strictly versioned
Modern HAL	Post-Android 8 HIDL-based HAL with proper interfaces

---

Let me know if you'd like a visual diagram or real code examples to compare **Legacy HAL vs. HIDL HAL**.

67. how hal talk with driver

Great question! Let's break it down **simply and clearly**:

🧩 How does a HAL talk to a driver?

A **HAL (Hardware Abstraction Layer)** is a **C/C++ code layer** that sits **between the Android framework and the hardware drivers**.

It **does not directly control the hardware**, but instead **uses the Linux kernel drivers** underneath to do so.

🔄 The Communication Flow:

```scss
Android Framework (Java)
        ↓
    HAL (C/C++)
        ↓
 Linux Kernel Driver (C)
        ↓
   Actual Hardware
```

🛠️ How HAL calls driver code?

Inside the HAL C/C++ code, the HAL developer will:

✅ Use standard Linux system calls like:

- **`open()`**, **`read()`**, **`write()`**, **`ioctl()`**
  to **communicate with device drivers**, just like any user-space C program.

Example:

```c
int fd = open("/dev/some_device", O_RDWR);
ioctl(fd, SOME_IOCTL_COMMAND, &data);
```

- The device node (**`/dev/some_device`**) is created by the **Linux kernel** for a driver.
- These calls send data or commands to the **kernel driver**.

✅ Use custom ioctl commands:

- **`ioctl()`** lets HAL send **custom instructions** to the kernel driver.

Example:

```c
#define MY_IOCTL_CMD 0x1001
ioctl(fd, MY_IOCTL_CMD, &my_data);
```

- This tells the kernel driver to perform some action (e.g. "turn on the LED").

📦 Where are drivers?

- Android drivers are just **Linux kernel drivers**.
- They live in the kernel tree, like:
  `/drivers/some_subsystem/`

or for vendor drivers:
`/vendor/.../drivers/`

🧠 Example: Camera HAL → Driver

1. **Camera app** uses `CameraManager` (Java).
2. Android framework talks to **camera HAL**.
3. HAL calls something like:

```c
int fd = open("/dev/video0", O_RDWR);
ioctl(fd, VIDIOC_STREAMON, &args);
```

4. `/dev/video0` is handled by the **V4L2 camera driver**.
5. The **kernel driver** controls the camera hardware.

🔒 Notes:

- HAL talks to drivers in **user space**.
- Drivers run in **kernel space**.
- They communicate via the `/dev/*` **interfaces**, **sysfs**, or **ioctl** calls.
- Android does **not force** a strict way of doing this; HAL writers can choose how to use the driver.

🔁 Summary

Layer	Role	Language
Android HAL	Talks to drivers	C/C++
Linux Driver	Talks to actual hardware	C (kernel)
Interface	`/dev/` node + ioctl/syscalls	

Let me know if you want a **real code sample** of HAL calling a driver via `open/ioctl`.

68. Changes in hidl over the years

Here's a **clear summary of how HIDL evolved over the years** and what changed in each Android version:

🔁 Background: What is HIDL?

HIDL (HAL Interface Definition Language) was introduced in **Android 8.0 (Oreo)** to provide:

- **Stable, versioned HAL interfaces**
- **IPC over Binder**
- Separation between Android **framework** and **vendor code**

It replaced the older **legacy HALs** (pre-Android 8).

📜 Timeline of HIDL Changes

✅ Android 8.0 (Oreo) – HIDL Introduced

- Major re-architecture.
- **HIDL introduced as the new HAL format**
- HALs became **versioned**: `@1.0`, `@1.1`, etc.
- Code auto-generated for both **C++ and Java**.
- Supported features:
  - `interface`, `struct`, `enum`, `vec`, `safe_union`
  - Transport via **Binderized** or **Passthrough**
  - Tools like `hidl-gen` and `vts-tradefed`

✅ Android 9 (Pie)

- **Lazy HALs** supported:
  - HALs can shut down and restart dynamically.
  - Uses `LazyServiceRegistrar` instead of `registerAsService()`.
- **Dynamic HAL loading** via `interface` keyword in `init.rc`
- Support for new annotations like `@callflow`
- HAL lifecycle improvements

✅ Android 10 (Q)

- **HIDL Java interface moved out of bootclasspath**
  - Required shallow Java libs + `jarjar` rules
- **Shallow Java libraries** introduced (`-shallow` variants)
- Enhanced **death recipient** handling
- Expanded support for `handle` and `memory` types in Java
- Use of **FMQ** (Fast Message Queue) for low-latency data exchange
- **VINTF validation** required for HIDL declarations

✅ Android 11 (R)

- **`hidl_memory` and `handle` support** added in Java

- Improvements to FMQ usage
- **Structured data types** in safe unions and nested structs
- HIDL fuzzing support via **libhidl-gen-fuzzer**
- HIDL interfaces could now be tested using VTS-style fuzzing

---

✅ Android 12 (S)

- **Migration from HIDL to AIDL began**:
  - New HALs must use **AIDL** (preferred now).
  - AIDL supports **versioning**, **structured types**, and **stronger guarantees**.
- HIDL still maintained for backward compatibility.

✅ Android 13–14

- No new HIDL features.
- **Strong migration push to AIDL**.
- AIDL supports Java, NDK, Rust, C++
- HIDL considered **deprecated for new development**.

🔴 Android 15+ (expected/future)

- **Only AIDL** HALs for new subsystems.
- **HIDL HALs must still work**, but discouraged.
- Official support and tooling for HIDL may be removed in later releases.

🧾 Summary Table

Android Version	HIDL Changes / Notes
8.0 Oreo	HIDL introduced, replaces legacy HALs
9.0 Pie	Lazy HALs, dynamic lifecycle via init.rc
10 Q	Java support refined, handle/memory supported, FMQ expanded
11 R	Java **`handle`**/**`memory`** support, safe_union, fuzzing
12 S	**AIDL recommended**; new HALs must use AIDL
13+	No new HIDL features; focus on AIDL only

🚨 Final Note:

- **Do not write new HALs using HIDL.**
- Use **AIDL (Android Interface Definition Language)** instead.
- HIDL is only maintained for legacy HALs like **`android.hardware.camera@3.4`**.

Would you like a **comparison between HIDL and AIDL** next?

69. Yes

Here's a clear and concise **comparison between HIDL and AIDL** in the context of Android HAL development:

📊 HIDL vs AIDL (Android Interface Definition Language)

Feature / Aspect	HIDL (HAL Interface Definition Language)	AIDL (Android Interface Definition Language)
📅 Introduced In	Android 8.0 (Oreo)	Android (pre-Oreo), but extended for HALs in Android 10+
🧱 Primary Use	HAL development (native, versioned IPC)	HALs (from Android 10+), system services, inter-process communication
⚙️ Language Support	**C++**, Java (limited), no Rust	**Java**, **C++ (NDK)**, **Rust**, AIDL backend
🧩 Interface Versioning	Fully supported: `@1.0`, `@1.1`, `@2.0`, etc.	Supported via `.aidl` `@version` + `@vintf` metadata
🧬 Type Support	`int`, `string`, `vec`, `struct`, `enum`, `safe_union`, `fmq`, `hidl_memory`, `handle`	`int`, `string`, `List`, `Parcelable`, `enum`, `union`, `FileDescriptor`, etc.
📚 Tooling	`hidl-gen`, `vts-tradefed`, manual integration	`aidl_interface`, `aidl-cpp`, `aidl-rust`, fully integrated
🔌 Transport	Binder (passthrough or binderized)	Always **Binderized**
🔄 Backward Compatibility	Versioned APIs allow backward compatibility	Versioned APIs with **stability guarantees** via VINTF
📦 Code Generation	Generates **C++/Java Stub & Proxy** classes	Generates **Stub & Proxy** for each language backend
🚀 Performance	Good (supports FMQ for fast data)	Good; **NDK backend** supports zero-copy, binder pooling
🔁 Service Lifecycle	Supports **lazy HALs** with `LazyServiceRegistrar`	Same (with automatic management via AIDL framework)
🧪 Testing Support	Supported via **VTS** and `hidl-gen-fuzzer`	Supported via **vts-tradefed**, **fuzzing**, and **mocking**
🚫 Deprecated	**Yes** (for new HALs since Android 10)	**No** – Officially recommended for all new HALs

✅ When to Use AIDL

Google requires **AIDL for all new HALs** (Android 10+), because:

- It supports **multi-language** (C++, Java, Rust)
- Provides better **testing**, **versioning**, and **tooling**
- Easier to integrate with **system services**
- Better suited for **modern HAL development**

❌ When to Use HIDL

You **should not** use HIDL unless:

- You're maintaining an **existing HIDL HAL**
- The HAL was originally written in HIDL (e.g., `android.hardware.camera@3.5`)

🔁 Quick Syntax Comparison

HIDL (interface):

```hidl
interface IFoo {
    doSomething(int32_t x) generates (int32_t y);
};
```

AIDL (interface):

```aidl
interface IFoo {
    int doSomething(int x);
```

```
}
```

🏁 Summary

Decision	Use
Writing new HALs (Android 10+)	✅ **AIDL**
Maintaining existing legacy HALs	⚠️ **HIDL**
Writing services in Java or Rust	✅ **AIDL**
Need fine-grained versioning across HALs	✅ Both support, but AIDL is cleaner

Would you like a **full working AIDL HAL example**, or a guide on **migrating from HIDL to AIDL** next?

70. Why hidl deprecated give concise answer

HIDL was deprecated because:

- ✅ **AIDL is simpler and unified**: AIDL works across Java, C++, and Rust, reducing complexity.
- ✅ **Better tooling**: AIDL has stronger integration with Android build, testing (VTS), and fuzzing tools.
- ✅ **Improved stability & versioning**: AIDL provides clearer, safer backward/forward compatibility with `@version`, `@stable`, etc.
- ✅ **Easier HAL development**: AIDL removes the need for custom `hidl-gen` and binder setup.

📌 **Summary**: AIDL replaced HIDL to simplify HAL development, unify system interfaces, and improve maintainability across Android.

71. heserivemanager vs servicemanager vs vndservicemanager

Here's a **concise comparison** of `hwservicemanager`, `servicemanager`, and `vndservicemanager` in Android:

Component	Purpose	Used by	Interface Type	Lives in
`servicemanager`	Manages **Java AIDL** services (System services)	Framework (Java)	**Java AIDL**	`system_server`
`hwservicemanager`	Manages **HIDL** services	HALs (C++)	**HIDL**	`init` process
`vndservicemanager`	Manages **Legacy Binder IPC** for vendor processes (pre-AIDL HALs)	Vendor-side services	**Binder (C++)**	`vendor`

🧩 In More Detail:

✅ `servicemanager`

- Manages classic **Java AIDL** services (like `activity`, `package`, etc.).
- Accessed using `defaultServiceManager()`.
- Framework code (Java/Kotlin) registers services here.

✅ `hwservicemanager`

- Manages **HIDL** interfaces.
- Used to register and look up HALs in the `/vendor` or `/system`.
- Accessed via `getService()` in HIDL.
- Introduced in Android 8 for modular HALs.

✅ `vndservicemanager`

- Vendor-only service manager.
- Used **only for legacy vendor services** using C++ Binder.
- Deprecated and rarely used after Android 10.

🚨 Which to Use?

- Use **AIDL + servicemanager** for new development.
- Use **hwservicemanager** only if you're still on HIDL-based HALs.
- **Don't use** `vndservicemanager` in new code—it's obsolete.

Let me know if you want a diagram of how they connect.