AIDL

1. AIDL overview

bookmark_border
The Android Interface Definition Language (AIDL) is a tool that lets users abstract away IPC. Given an interface (specified in a .aidl file), various build systems use the aidl binary to construct C++ or Java bindings so that this interface can be used across processes, regardless of the runtime or bitness there.

AIDL can be used between any process in Android: between platform components or between apps. However, it is never used as an API for apps. AIDL may be used to implement an SDK API in the platform, for example, but the SDK API surface never contains AIDL APIs directly. For documentation about how to use AIDL between apps directly, see corresponding Android developers documentation. When AIDL is used between platform components that are updated separately, such as APEXes (starting in Android 10) or HALs (starting in Android 11), the versioning system known as Stable AIDL must be used.

Example
Here is an example AIDL interface:

```
package my.package;

import my.package.Baz; // defined elsewhere

interface IFoo {
    void doFoo(Baz baz);
}
```
A server process registers an interface and serves calls to it, and a client process makes calls to those interfaces. In many cases, a process acts as both a client and a server since it may be referencing multiple interfaces. For more details about the AIDL language, see AIDL language. For more details about the various runtimes available to use these interfaces, see AIDL backends. These type declarations are exactly like a class declaration in a given language, but they work across processes.

How it works
AIDL uses the binder kernel driver to make calls. When you make a call, a method identifier and all of the objects are packed onto a buffer and copied to a remote process where a binder thread waits to read the data. Once a binder thread receives data for a transaction, the thread looks up a native stub object in the local process, and this class unpacks the data and makes a call on a local interface object. This local interface object is the one a server process creates and registers. When calls are made in the same process and the same backend, no proxy objects exist, and so calls are direct without any packing or unpacking.

Interact with services on the device
Android comes with a few commands to allow interacting with services on the device. Try:

```
adb shell dumpsys --help # listing and dumping services
adb shell service --help # sending commands to services for testing
```

Here's a simplified **overview of AIDL (Android Interface Definition Language)**, breaking down each part for clarity:

🔧 What is AIDL?

- **AIDL** is a tool in Android that helps different **processes communicate** with each other using **IPC (Inter-Process Communication)**.
- It's especially used when data needs to be sent between separate applications or between **Android system components** (like system services or HALs).
- It **hides the complexity** of binder IPC behind a language-level interface, so developers can just define the interface, and the system takes care of the rest.

💡 When & Where Is AIDL Used?

- **Platform-to-Platform IPC**: e.g., between system services, HALs, or APEX modules.
- **App-to-App or App-to-System (rare)**: Usually discouraged unless tightly controlled (not exposed in SDK APIs).
- **HALs or APEX Modules**: If those use AIDL, they must use **Stable AIDL** for versioning and backward compatibility.

⚠️ AIDL interfaces **are not directly part of the Android public SDK API**.

🧪 Example AIDL Interface

```aidl
package my.package;

import my.package.Baz; // another AIDL-defined type

interface IFoo {
    void doFoo(Baz baz);
}
```

- This defines an interface **`IFoo`** that has a method **`doFoo`** accepting a **`Baz`** object.
- **`Baz`** must be another AIDL-defined parcelable type.

🧠 How Does AIDL Work Internally?

1. **Interface Compilation**:
    - The **`.aidl`** file is compiled into **Java/C++/Rust stub and proxy classes**.
    - These classes handle IPC details like marshaling (packing) and unmarshaling (unpacking) data.
2. **Runtime Communication**:
    - A **server process** creates an implementation of the interface and **registers** it with the Android **Binder** driver.
    - A **client process** obtains a proxy object to the interface and makes method calls on it.
    - Binder copies the data across process boundaries, invokes the server-side method, and returns the result.
3. **Optimization**:
    - If client and server are in the **same process**, the proxy is bypassed and calls are **direct**.

⚙️ Commands to Interact with AIDL Services (via Binder)

- **Check system services**:

  ```bash
  adb shell dumpsys --help
  ```

  (Use to inspect running services)
- **Send binder calls directly for testing**:

  ```bash
  adb shell service --help
  ```

  (Used for low-level binder call testing)

🧭 AIDL vs Other IPC Methods in Android

IPC Mechanism	Use Case	Notes
AIDL	Complex IPC with types/callbacks	Best for platform or HALs
Messenger	IPC using **`Handler`** and **`Message`**	Simplified, good for app-to-service
Intents	IPC using broadcasts or startService	Good for loosely coupled communication
ContentProvider	Structured data access across apps	Best for sharing data like a DB

📚 Next Topics to Explore

- **`Stable AIDL`** (for versioning)
- **`AIDL language syntax`** and types (e.g., **`in`**, **`out`**, **`inout`**)

- `Backends`: Java, C++, Rust support
- Service registration: `IServiceManager`, `ServiceManager.addService()`

Would you like a **full working example** with `IFoo.aidl`, a client app, and a service app using AIDL?

2. AIDL language

bookmark_border
The AIDL language is loosely based on the Java language. Files specify an interface contract and various data types and constants used in this contract.

Package
Every AIDL files starts with an optional package that corresponds to the package names in various backends. A package declaration looks like this:

```
package my.package;
```

Similar to Java, AIDL files must be in a folder structure matching their package. Files with package my.package must be in the folder my/package/.

Types
In AIDL files, there are many places where types can be specified. For an exact list of types that are supported in the AIDL language, see AIDL backends types.

Annotations
Several parts of the AIDL language support annotations. For a list of annotations and where they can be applied, see AIDL annotations.

Imports
To use types defined in other interfaces, you must first add dependencies in the build system. In cc_* and java_* Soong modules, where .aidl files are used directly under srcs in Android platform builds, you can add directories using the field aidl: { include_dirs: ... }. For imports using aidl_interface, see here.

An import looks like this:

```
import some.package.Foo;  // explicit import
```

When importing a type in the same package, the package can be omitted. Though, omitting the package can lead to ambiguous import errors when types are specified without a package and put in the global namespace (generally all types should be namespaced):

```
import Foo;  // same as my.package.Foo
```

Define types
AIDL files generally define types which are used as an interface.

Interfaces
Here is an example AIDL interface:

```
interface ITeleport {
    // Location defined elsewhere
    void teleport(Location baz, float speed);
    String getName();

    // ITeleportCallback defined elsewhere
    void methodWithCallback(ITeleportCallback callback);

    // ITeleportSession defined elsewhere
    ITeleportSession getASubInterface();
}
```

An interface defines an object with a series of methods. Methods can either be oneway (oneway void doFoo()) or synchronous. If an interface is defined as oneway (oneway interface ITeleport {...}), then all methods in it are implicitly oneway. Oneway methods are dispatched asynchronously and cannot return a result. One-way methods from the same thread to the same binder also execute serially (though potentially on different threads). For a discussion of how to setup threads, see AIDL backends thread management.

Binder allows many interfaces and binder objects to be shared through binder interfaces. AIDL interfaces frequently employ callbacks as part of method calls, such as with ITeleportCallback in the previous example. You can reuse callback objects between calls to the same method or calls to different methods. Another common use of interface types is for sub-interfaces or session objects to be returned from methods such as with ITeleportSession in the previous example. This nesting allows different APIs to be encapsulated at either the API or based on runtime state. For example, a session may represent ownership of a particular resource. When interfaces are passed around multiple times or returned to the client or server they came from, they always preserve pointer equality of the underlying binder object.

Methods can have zero or more arguments. Arguments to methods can be in, out, or inout. For a discussion of how this affects arguments types, see AIDL backends directionality.

Parcelables
For a description of how to create backend-specific parcelables, AIDL backends custom parcelables.

Android 10 and higher support parcelable definitions directly in AIDL. This type of parcelable is called a structured parcelable. For further information on how structured and stable AIDL are related in the AIDL compiler and our build system, see Structured versus stable AIDL.

For example:

```
package my.package;

import my.package.Boo;

parcelable Baz {
    @utf8InCpp String name = "baz";
    Boo boo;
}
```

Unions
Android 12 and higher support tagged union declarations. For example:

```
package my.package;

import my.package.FooSettings;
import my.package.BarSettings;

union Settings {
    FooSettings fooSettings;
    BarSettings barSettings;
    @utf8InCpp String str;
    int number;
}
```

See AIDL Backends Unions for backend-specific details.

Enums
Android 11 and higher support enum declarations. For example:

```
package my.package;

enum Boo {
    A = 1 * 4,
    B = 3,
}
```

Nested Type Declarations
Android 13 and higher support nested type declarations. For example:

```
package my.package;

import my.package.Baz;

interface IFoo {
    void doFoo(Baz.Nested nested);  // defined in my/package/Baz.aidl
    void doBar(Bar bar);            // defined below

    parcelable Bar { ... }          // nested type definition
}
```

Constants
Custom AIDL interfaces, parcelables, and unions can also contain integer and string constants, such as:

const @utf8InCpp String HAPPY = ":)";
const String SAD = ":(";
const byte BYTE_ME = 1;
const int ANSWER = 6 * 7;
Constant expressions
AIDL constants, array sizes, and enumerators can be specified using constant expressions. Expressions can use parentheses to nest operations. Constant expression values can be used with integral or float values.

true and false literals represent boolean values. Values with a . but without a suffix, such as 3.8, are considered to be double values. Float values have the f suffix, such as 2.4f. An integral value with the l or L suffix indicates a 64-bit long value. Otherwise, integrals values get the smallest value-preserving signed type between 8-bit (byte), 32-bit (int), and 64-bit (long). So 256 is considered to be an int, but 255 + 1 overflows to be the byte 0. Hex values, such as 0x3, are first interpreted as the smallest value-preserving unsigned type between 32-bit and 64-bit and then reinterpreted as unsigned values. So, 0xffffffff has the int value -1. Starting in Android 13, the suffix u8 can be added to constants, such as 3u8, to represent a byte value. This suffix is important so that a calculation, such as 0xffu8 * 3, is interpreted as -3 with type byte whereas 0xff * 3 is 765 with type int.

Supported operators have C++ and Java semantics. In order from lowest to highest precedence, binary operators are || && | ^ & == != < > <= >= << >> + - * / %. Unary operators are + - ! ~.

Was this helpful?

Yes, that was a helpful and comprehensive explanation of the **AIDL language**. Here's a **clean summary with intuitive structure and examples** to make everything clearer for you:

✅ AIDL Language Overview

AIDL (Android Interface Definition Language) is a language similar to Java used to define interfaces for **IPC (Inter-Process Communication)**. These interfaces are used by the Android binder framework to allow different processes to communicate.

📦 1. Package Declaration

Like Java:

```aidl
package my.package;
```

- Must match directory structure: `my/package/`

📂 2. Imports

Used to bring in other AIDL-defined types:

```aidl
import my.package.Foo;       // full path
import Foo;                  // same package (not recommended)
```

> Add correct `include_dirs` in `aidl_interface` or `cc_/java_` modules for build system to find files.

🔧 3. Interfaces

Defines a set of methods that can be invoked across processes:

```aidl
interface ITeleport {
    void teleport(Location loc, float speed);
    String getName();
    void methodWithCallback(ITeleportCallback callback);
    ITeleportSession getASubInterface();
}
```

🔁 Method Types:

- **`void foo()`** → synchronous
- **`oneway void foo()`** → asynchronous (no return, queued)

🧭 **Parameter Direction:**

- **`in`**: passed from caller to callee
- **`out`**: callee modifies and returns it
- **`inout`**: both directions

📦 4. Parcelables

Used to define complex data types:

```aidl
parcelable Baz {
    @utf8InCpp String name = "baz";
    Boo boo;  // another parcelable
}
```

> Introduced as "structured parcelables" in Android 10+

🧊 5. Unions (Android 12+)

Only **one field can be set at a time**:

```aidl
union Settings {
    FooSettings fooSettings;
    BarSettings barSettings;
    @utf8InCpp String str;
    int number;
}
```

🔢 6. Enums (Android 11+)

```aidl
enum Boo {
    A = 1 * 4,
    B = 3,
}
```

🧩 7. Nested Type Declarations (Android 13+)

```aidl
interface IFoo {
    void doFoo(Baz.Nested nested);  // external nested type
    void doBar(Bar bar);            // local nested type

    parcelable Bar {
        int id;
        String desc;
    }
}
```

🔐 8. Constants

Can be used inside **`interface`**, **`parcelable`**, **`union`**:

```aidl
const @utf8InCpp String HAPPY = ":)";
const String SAD = ":(";
```

```
const byte BYTE_ME = 1;
const int ANSWER = 6 * 7;
```

🧮 9. Constant Expressions

- Types: `int`, `byte`, `long`, `float`, `double`
- Literals:
  - Boolean: `true`, `false`
  - Hex: `0x3`, `0xff`
  - Float: `2.4f`
  - Long: `1234L`
  - Byte: `3u8` (Android 13+)

Operators (like C++/Java):

- Binary: `||`, `&&`, `==`, `!=`, `+`, `-`, `*`, `/`, `%`, `<<`, `>>`
- Unary: `+`, `-`, `!`, `~`

📌 Summary Table

Feature	Introduced in	Example Syntax
Package	—	`package my.package;`
Interface	—	`interface IFoo {...}`
Parcelables	Android 10	`parcelable Foo { ... }`
Unions	Android 12	`union U { int x; String y;}`
Enums	Android 11	`enum Color { RED, GREEN }`
Constants	—	`const int MAX = 10;`
Nested Types	Android 13	`parcelable Bar {}` inside `interface`

Would you like a complete **working AIDL example with client and service apps** using these language features?

3. ↪
🧭 Parameter Direction: * `in`: passed from caller to callee * `out`: callee modifies and returns it * `inout`: both directions
explain this in simple with example

Sure! Let's break down the AIDL **parameter direction types** (`in`, `out`, `inout`) in **very simple terms** using a **real-life analogy** and a **code example**.

🧭 What are `in`, `out`, and `inout`?

They describe **how data flows** between the **client** and the **service** (server):

Direction	Who Sends	Who Receives	Editable by Service?	Real-world analogy
`in`	Client	Service	✅ Yes	You send a form to someone.
`out`	Service	Client	✅ Yes	You send an **empty** form and get it back filled.
`inout`	Client	Both	✅ Yes	You send a **half-filled** form and get it back updated.

🧪 Let's say we have a parcelable:

```
aidl

parcelable Person {
    String name;
    int age;
}
```

🧑‍💻 AIDL Interface with all three:

```
aidl

interface ITest {
    void sendInPerson(in Person person);       // Send only
    void getOutPerson(out Person person);      // Get result
    void updateInOutPerson(inout Person person); // Send & Get update
}
```

💬 How this works:

1. `in Person person`

- Client sends: `Person("Alice", 30)`
- Server receives the data.
- Any changes on server **don't come back** to client.

```
java

// Client side
Person p = new Person("Alice", 30);
testService.sendInPerson(p);
// p is still "Alice", 30 after call
```

---

2. `out Person person`

- Client sends: **an empty object**
- Server fills it up.

```java
// Client side
Person p = new Person(); // blank
testService.getOutPerson(p);
// p is now filled, e.g., "Bob", 45
```

---

3. `inout Person person`

- Client sends: `Person("Charlie", 25)`
- Server updates the object, e.g., change name and age
- Client sees the changes

```java
// Client side
Person p = new Person("Charlie", 25);
testService.updateInOutPerson(p);
// p is now updated, e.g., "Charlie", 26
```

📦 Server-side example in Java:

```java
@Override
public void updateInOutPerson(Person person) {
    person.age += 1;              // Add 1 to age
    person.name = "Updated " + person.name;
}
```

✅ Summary

Direction	Sent to Service	Returned from Service	Use When...
`in`	✅	❌	You want to **send only**
`out`	❌ (empty)	✅	You want to **receive only**
`inout`	✅	✅	You want to **send & receive**

Would you like me to give you the **full working example in Android with AIDL service and client** for these 3 cases?

4. AIDL backends

bookmark_border
An AIDL backend is a target for stub code generation. Always use AIDL files in a particular language with a specific runtime. Depending on the context, you should use different AIDL backends.

In the following table, the stability of the API surface refers to the ability to compile code against this API surface in a way that the code can be delivered independently from the system.img libbinder.so binary.

AIDL has the following backends:

Backend	Language	API surface	Build systems
Java	Java	SDK or SystemApi (stable*)	All
NDK	C++	libbinder_ndk (stable*)	aidl_interface
CPP	C++	libbinder (unstable)	All
Rust	Rust	libbinder_rs (stable*)	aidl_interface

These API surfaces are stable, but many of the APIs, such as those for service management, are reserved for internal platform use and aren't available to apps. For more information on how to use AIDL in apps, see Android Interface Definition Language (AIDL).
The Rust backend was introduced in Android 12; the NDK backend has been available as of Android 10.
The Rust crate is built on top of libbinder_ndk, which lets it be stable and portable. APEXes use the binder crate in the standard way on the system side. The Rust portion is bundled into an APEX and shipped inside it. This portion depends on the libbinder_ndk.so on the system partition.
Build systems
Depending on the backend, there are two ways to compile AIDL into stub code. For more details on the build systems, see Soong Modules Reference.

Core build system
In any cc_ or java_ Android.bp module (or in their Android.mk equivalents), you can specify AIDL (.aidl) files as source files. In this case, the Java or CPP backends of AIDL are used (not the NDK backend), and the classes to use the corresponding AIDL files are added to the module automatically. You can specify options such as local_include_dirs (which tells the build system the root path to AIDL files in that module) in these modules under an aidl: group.

The Rust backend is only for use with Rust. rust_ modules are handled differently in that AIDL files aren't specified as source files. Instead, the aidl_interface module produces a rustlib called aidl_interface_name-rust, which can be linked against. For details, see the Rust AIDL example.

Warning: In Android 13 and lower, the NDK backend compiles only with weak symbols and requires __ANDROID_UNAVAILABLE_SYMBOLS_ARE_WEAK__ to be set. In the Android platform build system, this is set by default.
Warning: libbinder_ndk has two sets of headers. For platform devs, these are shipped together, but if you're an app developer, you need to include (usually with -I for C++ compilers, such as clang) them from different places. The core headers for APIs that are provided on device (such as android/binder_ibinder.h, defined in the Android tree at frameworks/native/libs/binder/ndk/include_ndk) are shipped with the NDK. However, other helper files (such as android/binder_interface_utils.h, defined in the Android tree frameworks/native/libs/binder/ndk/include_cpp) provide C++ utilities that are required to use with the AIDL compiler and shipped with it in the SDK under platforms/$PLATFORM_NAME/optional/libbinder_ndk_cpp/.
aidl_interface
Types used with the aidl_interface build system must be structured. In order to be structured, parcelables must contain fields directly and not be declarations of types defined directly in target languages. For how structured AIDL fits in with stable AIDL, see Structured versus stable AIDL.

Types
Consider the aidl compiler as a reference implementation for types. When you create an interface, invoke aidl --lang=<backend> ... to see the resulting interface file. When you use the aidl_interface module, you can view the output in out/soong/.intermediates/<path to module>/.

Java or AIDL type	C++ type	NDK type	Rust type
boolean	bool	bool	bool
byte8	int8_t	int8_t	i8
char	char16_t	char16_t	u16
int	int32_t	int32_t	i32
long	int64_t	int64_t	i64
float	float	float	f32
double	double	double	f64
String	android::String16	std::string	In: &str Out: String
android.os.Parcelable	android::Parcelable	N/A	N/A
IBinder	android::IBinder	ndk::SpAIBinder	binder::SpIBinder
T[]	std::vector<T>	std::vector<T>	In: &[T]

Out: Vec<T>
byte[] std::vector std::vector1 In: &[u8]
Out: Vec<u8>
List<T> std::vector<T>2 std::vector<T>3 In: In: &[T]4
Out: Vec<T>
FileDescriptor android::base::unique_fd N/A N/A
ParcelFileDescriptor android::os::ParcelFileDescriptor ndk::ScopedFileDescriptor binder::parcel::ParcelFileDescriptor
Interface type (T) android::sp<T> std::shared_ptr<T>7 binder::Strong
Parcelable type (T) T T T
Union type (T)5 T T T
T[N]6 std::array<T, N> std::array<T, N> [T; N]

1. In Android 12 or higher, byte arrays use uint8_t instead of int8_t for compatibility reasons.

2. The C++ backend supports List<T> where T is one of String, IBinder, ParcelFileDescriptor or parcelable. In Android 13 or higher, T can be any nonprimitive type (including interface types) except arrays. AOSP recommends using array types like T[], because they work in all backends.

3. The NDK backend supports List<T> where T is one of String, ParcelFileDescriptor or parcelable. In Android 13 or higher, T can be any nonprimitive type except arrays.

4. Types are passed differently for Rust code depending on whether they are input (an argument), or an output (a returned value).

5. Union types are supported in Android 12 and higher.

6. In Android 13 or higher, fixed-size arrays are supported. Fixed-size arrays can have multiple dimensions (for example, int[3][4]). In the Java backend, fixed-size arrays are represented as array types.

7. To instantiate a binder SharedRefBase object, use SharedRefBase::make\<My\>(... args ...). This function creates a std::shared_ptr\<T\> object, which is also managed internally, in case the binder is owned by another process. Creating the object other ways causes double ownership.

8. See also Java or AIDL type byte[].

Directionality (in, out, and inout)
When specifying the types of the arguments to functions, you can specify them as in, out, or inout. This controls the direction that information is passed for an IPC call.

The in argument specifier indicates data is passed from the caller to the callee. The in specifier is the default direction, but if data types can also be out, then you must specify the direction.

The out argument specifier means that data is passed from the callee to the caller.

The inout argument specifier is the combination of both of these. However, we recommend avoiding using the argument specifier inout. If you use inout with a versioned interface and an older callee, the additional fields that are present only in the caller get reset to their default values. With respect to Rust, a normal inout type receives &mut T, and a list inout type receives &mut Vec<T>.

```
interface IRepeatExamples {
    MyParcelable RepeatParcelable(MyParcelable token); // implicitly 'in'
    MyParcelable RepeatParcelableWithIn(in MyParcelable token);
    void RepeatParcelableWithInAndOut(in MyParcelable param, out MyParcelable result);
    void RepeatParcelableWithInOut(inout MyParcelable param);
}
```

UTF-8 and UTF-16
With the CPP backend, you can choose whether strings are UTF-8 or UTF-16. Declare strings as @utf8InCpp String in AIDL to automatically convert them to UTF-8. The NDK and Rust backends always use UTF-8 strings. For more information about the utf8InCpp annotation, see utf8InCpp.

Nullability
You can annotate types that can be null with @nullable. For more information about the nullable annotation, see nullable.

Warning: Guard against unexpected values. Even if your client or server always passes a nonnull value, input must be checked in case of a malicious client or in some cases, for values that become null. For example, in the NDK backend an interface type might appear as null if its host process quits during a transaction.
Custom parcelables
A custom parcelable is a parcelable that's implemented manually in a target backend. Use custom parcelables only when you're trying to add support to other languages for an existing custom parcelable which can't be changed.

Warning: Custom parcelables are error prone. When creating a new parcelable, use structured AIDL parcelables. Custom parcelables need to be

implemented in each backend separately, and they can be out of sync or have other issues.
Here's an example of an AIDL parcelable declaration:

```
package my.pack.age;
parcelable Foo;
```

By default, this declares a Java parcelable where my.pack.age.Foo is a Java class implementing the Parcelable interface.

For a declaration of a custom CPP backend parcelable in AIDL, use cpp_header:

```
package my.pack.age;
parcelable Foo cpp_header "my/pack/age/Foo.h";
```

The C++ implementation in my/pack/age/Foo.h looks like this:

```
#include <binder/Parcelable.h>

class MyCustomParcelable : public android::Parcelable {
public:
    status_t writeToParcel(Parcel* parcel) const override;
    status_t readFromParcel(const Parcel* parcel) override;

    std::string toString() const;
    friend bool operator==(const MyCustomParcelable& lhs, const MyCustomParcelable& rhs);
    friend bool operator!=(const MyCustomParcelable& lhs, const MyCustomParcelable& rhs);
};
```

For a declaration of a custom NDK parcelable in AIDL, use ndk_header:

```
package my.pack.age;
parcelable Foo ndk_header "android/pack/age/Foo.h";
```

The NDK implementation in android/pack/age/Foo.h looks like this:

```
#include <android/binder_parcel.h>

class MyCustomParcelable {
public:

    binder_status_t writeToParcel(AParcel* _Nonnull parcel) const;
    binder_status_t readFromParcel(const AParcel* _Nonnull parcel);

    std::string toString() const;

    friend bool operator==(const MyCustomParcelable& lhs, const MyCustomParcelable& rhs);
    friend bool operator!=(const MyCustomParcelable& lhs, const MyCustomParcelable& rhs);
};
```

In Android 15, for declaration of a custom Rust parcelable in AIDL, use rust_type:

```
package my.pack.age;
@RustOnlyStableParcelable parcelable Foo rust_type "rust_crate::Foo";
```

The Rust implementation in rust_crate/src/lib.rs looks like this:

```
use binder::{
    binder_impl::{BorrowedParcel, UnstructuredParcelable},
    impl_deserialize_for_unstructured_parcelable, impl_serialize_for_unstructured_parcelable,
    StatusCode,
};

#[derive(Clone, Debug, Eq, PartialEq)]
struct Foo {
    pub bar: String,
}
```

```
impl UnstructuredParcelable for Foo {
    fn write_to_parcel(&self, parcel: &mut BorrowedParcel) -> Result<(), StatusCode> {
        parcel.write(&self.bar)?;
        Ok(())
    }

    fn from_parcel(parcel: &BorrowedParcel) -> Result<Self, StatusCode> {
        let bar = parcel.read()?;
        Ok(Self { bar })
    }
}

impl_deserialize_for_unstructured_parcelable!(Foo);
impl_serialize_for_unstructured_parcelable!(Foo);
```

Then you can use this parcelable as a type in AIDL files, but it won't be generated by AIDL. Provide < and == operators for CPP and NDK backend custom parcelables to use them in union.

Warning: Custom parcelables in Java and C++ should be final. If a subclass of a parcelable is sent as a superclass, then the underlying format is mismatched. A parent class parcelable would need to encode how subclasses are written to a parcel. The easiest solution is to avoid subclassing parcelables. Use composition instead.

Default values

Structured parcelables can declare per-field default values for primitives, String fields, and arrays of these types.

```
    parcelable Foo {
      int numField = 42;
      String stringField = "string value";
      char charValue = 'a';
      ...
    }
```

In the Java backend, when default values are missing, fields are initialized as zero values for primitive types and null for nonprimitive types.

In other backends, fields are initialized with default initialized values when default values aren't defined. For example, in the C++ backend, String fields are initialized as an empty string and List<T> fields are initialized as an empty vector<T>. @nullable fields are initialized as null-value fields.

Unions

AIDL unions are tagged and their features are similar in all backends. They're constructed to the first field's default value and they have a language-specific way to interact with them:

```
    union Foo {
      int intField;
      long longField;
      String stringField;
      MyParcelable parcelableField;
      ...
    }
```

Java example

```
    Foo u = Foo.intField(42);              // construct

    if (u.getTag() == Foo.intField) {      // tag query
      // use u.getIntField()               // getter
    }

    u.setSringField("abc");                // setter
```

C++ and NDK example

```
    Foo u;                                 // default constructor

    assert (u.getTag() == Foo::intField);             // tag query
    assert (u.get<Foo::intField>() == 0);             // getter

    u.set<Foo::stringField>("abc");                   // setter
```

```
  assert (u == Foo::make<Foo::stringField>("abc")); // make<tag>(value)
```

Rust example

In Rust, unions are implemented as enums and don't have explicit getters and setters.

```
    let mut u = Foo::Default();            // default constructor
    match u {                              // tag match + get
      Foo::IntField(x) => assert!(x == 0);
      Foo::LongField(x) => panic!("Default constructed to first field");
      Foo::StringField(x) => panic!("Default constructed to first field");
      Foo::ParcelableField(x) => panic!("Default constructed to first field");
      ...
    }
    u = Foo::StringField("abc".to_string()); // set
```

Error handling

The Android OS provides built-in error types for services to use when reporting errors. These are used by binders and can be used by any services implementing a binder interface. Their use is well documented in the AIDL definition and they don't require any user-defined status or return type.

Output parameters with errors

When an AIDL function reports an error, the function might not initialize or modify output parameters. Specifically, output parameters might be modified if the error occurs during unparceling, as opposed to happening during the processing of the transaction itself. In general, when getting an error from an AIDL function, all inout and out parameters as well as the return value (which acts like an out parameter in some backends) should be considered to be in an indefinite state.

Which error values to use

Many of the built-in error values can be used in any AIDL interfaces, but some are treated in a special way. For example, EX_UNSUPPORTED_OPERATION and EX_ILLEGAL_ARGUMENT are OK to use when they describe the error condition, but EX_TRANSACTION_FAILED must not be used because it's treated specially by the underlying infrastructure. Check the backend specific definitions for more information on these built-in values.

If the AIDL interface requires additional error values that aren't covered by the built-in error types, they can use the special service-specific built-in error that allows the inclusion of a service-specific error value that's defined by the user. These service-specific errors are typically defined in the AIDL interface as a const int or int-backed enum and aren't parsed by binder.

In Java, errors map to exceptions, such as android.os.RemoteException. For service-specific exceptions, Java uses android.os.ServiceSpecificException along with the user-defined error.

Native code in Android doesn't use exceptions. The CPP backend uses android::binder::Status. The NDK backend uses ndk::ScopedAStatus. Every method generated by AIDL returns one of these, representing the status of the method. The Rust backend uses the same exception code values as the NDK, but converts them into native Rust errors (StatusCode, ExceptionCode) before delivering them to the user. For service-specific errors, the returned Status or ScopedAStatus uses EX_SERVICE_SPECIFIC along with the user-defined error.

The built-in error types can be found in the following files:

Backend	Definition
Java	android/os/Parcel.java
CPP	binder/Status.h
NDK	android/binder_status.h
Rust	android/binder_status.h

Use various backends

These instructions are specific to Android platform code. These examples use a defined type, my.package.IFoo. For instructions on how to use the Rust backend, see the Rust AIDL example in Android Rust patterns.

Import types

Whether the defined type is an interface, parcelable, or union, you can import it in Java:

```
import my.package.IFoo;
```

Or in the CPP backend:

```
#include <my/package/IFoo.h>
```

Or in the NDK backend (notice the extra aidl namespace):

```
#include <aidl/my/package/IFoo.h>
```

Or in the Rust backend:

```
use my_package::aidl::my::package::IFoo;
```

Although you can import a nested type in Java, in the CPP and NDK backends you must include the header for its root type. For example, when importing a nested type Bar defined in my/package/IFoo.aidl (IFoo is the root type of the file) you must include <my/package/IFoo.h> for the CPP backend (or <aidl/my/package/IFoo.h> for the NDK backend).

Implement an interface

To implement an interface, you must inherit from the native stub class. An implementation of an interface is often called a service when it's registered with the service manager or android.app.ActivityManager and called a callback when it's registered by a client of a service. However, a variety of names are used to describe interface implementations depending on the exact usage. The stub class reads commands from the binder driver and executes the methods that you implement. Imagine that you have an AIDL file like this:

```
package my.package;
interface IFoo {
    int doFoo();
}
```

In Java, you must extend from the generated Stub class:

```
import my.package.IFoo;
public class MyFoo extends IFoo.Stub {
    @Override
    int doFoo() { ... }
}
```

In the CPP backend:

```
#include <my/package/BnFoo.h>
class MyFoo : public my::package::BnFoo {
    android::binder::Status doFoo(int32_t* out) override;
}
```

In the NDK backend (notice the extra aidl namespace):

```
#include <aidl/my/package/BnFoo.h>
class MyFoo : public aidl::my::package::BnFoo {
    ndk::ScopedAStatus doFoo(int32_t* out) override;
}
```

In the Rust backend:

```
use aidl_interface_name::aidl::my::package::IFoo::{BnFoo, IFoo};
use binder;

/// This struct is defined to implement IRemoteService AIDL interface.
pub struct MyFoo;

impl Interface for MyFoo {}

impl IFoo for MyFoo {
    fn doFoo(&self) -> binder::Result<()> {
        ...
        Ok(())
    }
}
```

Or with async Rust:

```
use aidl_interface_name::aidl::my::package::IFoo::{BnFoo, IFooAsyncServer};
use binder;

/// This struct is defined to implement IRemoteService AIDL interface.
```

```
pub struct MyFoo;

impl Interface for MyFoo {}

#[async_trait]
impl IFooAsyncServer for MyFoo {
    async fn doFoo(&self) -> binder::Result<()> {
        ...
        Ok(())
    }
}
```

Register and get services

Services in platform Android are usually registered with the servicemanager process. In addition to the following APIs, some APIs check the service (meaning they return immediately if the service isn't available). Check the corresponding servicemanager interface for exact details. You can perform these operations only when compiling against platform Android.

Warning: AIDL or the binder doesn't start a threadpool by default. If waitForService is hanging, you might need to start the binder threadpool.

Warning: The getService method can retrieve services in Android. In the CPP, NDK, or Rust backend and some pieces of the Java backend, this method polls for 5 seconds and returns the service only if it starts within that window. Because getService is polling and can introduce races, we recommend avoiding this method.

In Java:

```
import android.os.ServiceManager;
// registering
ServiceManager.addService("service-name", myService);
// return if service is started now
myService = IFoo.Stub.asInterface(ServiceManager.checkService("service-name"));
// waiting until service comes up (new in Android 11)
myService = IFoo.Stub.asInterface(ServiceManager.waitForService("service-name"));
// waiting for declared (VINTF) service to come up (new in Android 11)
myService = IFoo.Stub.asInterface(ServiceManager.waitForDeclaredService("service-name"));
```

In the CPP backend:

```
#include <binder/IServiceManager.h>
// registering
defaultServiceManager()->addService(String16("service-name"), myService);
// return if service is started now
status_t err = checkService<IFoo>(String16("service-name"), &myService);
// waiting until service comes up (new in Android 11)
myService = waitForService<IFoo>(String16("service-name"));
// waiting for declared (VINTF) service to come up (new in Android 11)
myService = waitForDeclaredService<IFoo>(String16("service-name"));
```

In the NDK backend (notice the extra aidl namespace):

```
#include <android/binder_manager.h>
// registering
binder_exception_t err = AServiceManager_addService(myService->asBinder().get(), "service-name");
// return if service is started now
myService = IFoo::fromBinder(ndk::SpAIBinder(AServiceManager_checkService("service-name")));
// is a service declared in the VINTF manifest
// VINTF services have the type in the interface instance name.
bool isDeclared = AServiceManager_isDeclared("android.hardware.light.ILights/default");
// wait until a service is available (if isDeclared or you know it's available)
myService = IFoo::fromBinder(ndk::SpAIBinder(AServiceManager_waitForService("service-name")));
```

In the Rust backend:

Note: myfoo here is the crate name of the library containing the previously defined interface.

```
use myfoo::MyFoo;
use binder;
use aidl_interface_name::aidl::my::package::IFoo::BnFoo;

fn main() {
```

```
    binder::ProcessState::start_thread_pool();
    // [...]
    let my_service = MyFoo;
    let my_service_binder = BnFoo::new_binder(
        my_service,
        BinderFeatures::default(),
    );
    binder::add_service("myservice", my_service_binder).expect("Failed to register service?");
    // Does not return - spawn or perform any work you mean to do before this call.
    binder::ProcessState::join_thread_pool()
}
```

In the async Rust backend, with a single-threaded runtime:

```
use myfoo::MyFoo;
use binder;
use binder_tokio::TokioRuntime;
use aidl_interface_name::aidl::my::package::IFoo::BnFoo;

#[tokio::main(flavor = "current_thread")]
async fn main() {
    binder::ProcessState::start_thread_pool();
    // [...]
    let my_service = MyFoo;
    let my_service_binder = BnFoo::new_async_binder(
        my_service,
        TokioRuntime(Handle::current()),
        BinderFeatures::default(),
    );

    binder::add_service("myservice", my_service_binder).expect("Failed to register service?");

    // Sleeps forever, but does not join the binder threadpool.
    // Spawned tasks run on this thread.
    std::future::pending().await
}
```

One important difference from the other options is that you don't call join_thread_pool when using async Rust and a single-threaded runtime. This is because you need to give Tokio a thread where it can execute spawned tasks. In the following example, the main thread serves that purpose. Any tasks spawned using tokio::spawn execute on the main thread.

In the async Rust backend, with a multithreaded runtime:

```
use myfoo::MyFoo;
use binder;
use binder_tokio::TokioRuntime;
use aidl_interface_name::aidl::my::package::IFoo::BnFoo;

#[tokio::main(flavor = "multi_thread", worker_threads = 2)]
async fn main() {
    binder::ProcessState::start_thread_pool();
    // [...]
    let my_service = MyFoo;
    let my_service_binder = BnFoo::new_async_binder(
        my_service,
        TokioRuntime(Handle::current()),
        BinderFeatures::default(),
    );

    binder::add_service("myservice", my_service_binder).expect("Failed to register service?");

    // Sleep forever.
    tokio::task::block_in_place(|| {
        binder::ProcessState::join_thread_pool();
    });
}
```

With the multithreaded Tokio runtime, spawned tasks don't execute on the main thread. Therefore, it makes more sense to call join_thread_pool on the main thread so that the main thread isn't idle. You must wrap the call in block_in_place to leave the async context.

Link to death
You can request to get a notification for when a service hosting a binder dies. This can help to avoid leaking callback proxies or assist in error recovery. Make these calls on binder proxy objects.

In Java, use android.os.IBinder::linkToDeath.
In the CPP backend, use android::IBinder::linkToDeath.
In the NDK backend, use AIBinder_linkToDeath.
In the Rust backend, create a DeathRecipient object, then call my_binder.link_to_death(&mut my_death_recipient). Note that because DeathRecipient owns the callback, you must keep that object alive as long as you want to receive notifications.
Caller information
When receiving a kernel binder call, caller information is available in several APIs. The process ID (PID) refers to the Linux process ID of the process that's sending a transaction. The user ID (UI) refers to the Linux user ID. When receiving a one-way call, the calling PID is 0. Outside of a binder transaction context, these functions return the PID and UID of the current process.

Warning: PID is subject to races. As soon as you obtain a PID, the process containing the PID might die and another process might reuse the PID. Never assume that a PID represents a specific process. Generally, PIDs are useful only for debugging.
In the Java backend:

```
    ... = Binder.getCallingPid();
    ... = Binder.getCallingUid();
```

In the CPP backend:

```
    ... = IPCThreadState::self()->getCallingPid();
    ... = IPCThreadState::self()->getCallingUid();
```

In the NDK backend:

```
    ... = AIBinder_getCallingPid();
    ... = AIBinder_getCallingUid();
```

In the Rust backend, when implementing the interface, specify the following (instead of allowing it to default):

```
    ... = ThreadState::get_calling_pid();
    ... = ThreadState::get_calling_uid();
```

Bug reports and debugging API for services
When bug reports run (for example, with adb bugreport), they collect information from all around the system to aid with debugging various issues. For AIDL services, bug reports use the binary dumpsys on all services registered with the service manager to dump their information into the bug report. You can also use dumpsys on the command line to get information from a service with dumpsys SERVICE [ARGS]. In the C++ and Java backends, you can control the order in which services get dumped by using additional arguments to addService. You can also use dumpsys --pid SERVICE to get the PID of a service while debugging.

To add custom output to your service, override the dump method in your server object like you're implementing any other IPC method defined in an AIDL file. When doing this, restrict dumping to the app permission android.permission.DUMP or restrict dumping to specific UIDs.

In the Java backend:

```
    @Override
    protected void dump(@NonNull FileDescriptor fd, @NonNull PrintWriter fout,
        @Nullable String[] args) {...}
```

In the CPP backend:

```
    status_t dump(int, const android::android::Vector<android::String16>&) override;
```

In the NDK backend:

```
    binder_status_t dump(int fd, const char** args, uint32_t numArgs) override;
```

In the Rust backend, when implementing the interface, specify the following (instead of allowing it to default):

```
    fn dump(&self, mut file: &File, args: &[&CStr]) -> binder::Result<()>
```
Use weak pointers

You can hold a weak reference to a binder object.

Warning: In Android binder, you can promote a weak reference only if there is already a strong reference in the same process. If you only have a weak reference to a remote binder, you can't promote it.

While Java supports WeakReference, it doesn't support weak binder references at the native layer.

In the CPP backend, the weak type is wp<IFoo>.

In the NDK backend, use ScopedAIBinder_Weak:

```
#include <android/binder_auto_utils.h>

AIBinder* binder = ...;
ScopedAIBinder_Weak myWeakReference = ScopedAIBinder_Weak(AIBinder_Weak_new(binder));
```
In the Rust backend, use WpIBinder or Weak<IFoo>:

```
let weak_interface = myIface.downgrade();
let weak_binder = myIface.as_binder().downgrade();
```
Dynamically get interface descriptor

The interface descriptor identifies the type of an interface. This is useful when debugging or when you have an unknown binder.

In Java, you can get the interface descriptor with code such as:

```
    service = /* get ahold of service object */
    ... = service.asBinder().getInterfaceDescriptor();
```
In the CPP backend:

```
    service = /* get ahold of service object */
    ... = IInterface::asBinder(service)->getInterfaceDescriptor();
```
The NDK and Rust backends don't support this capability.

Statically get interface descriptor

Sometimes (such as when registering @VintfStability services), you need to know what the interface descriptor is statically. In Java, you can get the descriptor by adding code such as:

```
    import my.package.IFoo;
    ... IFoo.DESCRIPTOR
```
In the CPP backend:

```
    #include <my/package/BnFoo.h>
    ... my::package::BnFoo::descriptor
```
In the NDK backend (notice the extra aidl namespace):

```
    #include <aidl/my/package/BnFoo.h>
    ... aidl::my::package::BnFoo::descriptor
```
In the Rust backend:

```
    aidl::my::package::BnFoo::get_descriptor()
```
Enum range

In native backends, you can iterate over the possible values an enum can take on. Due to code size considerations, this isn't supported in Java.

For an enum MyEnum defined in AIDL, iteration is provided as follows.

In the CPP backend:

```
::android::enum_range<MyEnum>()
```

In the NDK backend:

```
::ndk::enum_range<MyEnum>()
```

In the Rust backend:

```
MyEnum::enum_values()
```

Thread management

Every instance of libbinder in a process maintains one threadpool. For most use cases, this should be exactly one threadpool, shared across all backends. The only exception is if vendor code loads another copy of libbinder to talk to /dev/vndbinder. This is on a separate binder node, so the threadpool isn't shared.

For the Java backend, the threadpool can only increase in size (because it's already started):

```
BinderInternal.setMaxThreads(<new larger value>);
```

For the CPP backend, the following operations are available:

```
// set max threadpool count (default is 15)
status_t err = ProcessState::self()->setThreadPoolMaxThreadCount(numThreads);
// create threadpool
ProcessState::self()->startThreadPool();
// add current thread to threadpool (adds thread to max thread count)
IPCThreadState::self()->joinThreadPool();
```

Similarly, in the NDK backend:

```
bool success = ABinderProcess_setThreadPoolMaxThreadCount(numThreads);
ABinderProcess_startThreadPool();
ABinderProcess_joinThreadPool();
```

In the Rust backend:

```
binder::ProcessState::start_thread_pool();
binder::add_service("myservice", my_service_binder).expect("Failed to register service?");
binder::ProcessState::join_thread_pool();
```

With the async Rust backend, you need two threadpools: binder and Tokio. This means that apps using async Rust need special considerations, especially when it comes to the use of join_thread_pool. See the section on registering services for more information on this.

Reserved names

C++, Java, and Rust reserve some names as keywords or for language-specific use. While AIDL doesn't enforce restrictions based on language rules, using field or type names that match a reserved name can result in a compilation failure for C++ or Java. For Rust, the field or type is renamed using the raw identifier syntax, accessible using the r# prefix.

We recommend avoiding using reserved names in your AIDL definitions where possible to avoid unergonomic bindings or outright compilation failure.

If you already have reserved names in your AIDL definitions, you can safely rename fields while remaining protocol compatible. You might need to update your code to continue building, but any already built programs continue to interoperate.

Names to avoid:

C++ keywords
Java keywords
Rust keywords
Was this helpful?

Here's a simplified explanation of **AIDL backends** and how to use them, with examples.

🔧 What is a Backend in AIDL?

A **backend** is the *target programming language* for which AIDL generates stub code. AIDL is just an interface definition — the backend determines how you implement and use it in Java, C++, Rust, etc.

📋 AIDL Backends Overview

Backend	Language	API Surface	Use With
Java	Java	Stable	Apps, Services
NDK	C++ (NDK-style)	Stable	Native system code, APEX
CPP	C++ (platform)	Unstable	Platform internals
Rust	Rust	Stable	System with Rust

> "Stable" means the interface remains compatible across updates. "Unstable" (CPP) is internal-only.

🚦 Example: Define a Common AIDL Interface

File: **`IFoo.aidl`**

```aidl
package my.package;

interface IFoo {
    int doSomething(int x);
}
```

1. ✅ Java Backend (for Apps or Services)

Implementation:

```java
public class FooImpl extends IFoo.Stub {
    @Override
    public int doSomething(int x) {
        return x * 2;
    }
}
```

Service Registration:

```java
ServiceManager.addService("foo", new FooImpl());
```

Client Usage:

```java
IFoo foo = IFoo.Stub.asInterface(ServiceManager.getService("foo"));
int result = foo.doSomething(10); // result = 20
```

2. ✅ C++ NDK Backend (for System Code/APEX)

Header file generated: `aidl/my/package/IFoo.h`

Implementation:

```cpp
#include <aidl/my/package/BnFoo.h>

class FooImpl : public aidl::my::package::BnFoo {
public:
    ndk::ScopedAStatus doSomething(int32_t x, int32_t* result) override {
        *result = x * 2;
        return ndk::ScopedAStatus::ok();
    }
};
```

Register:

```cpp
AServiceManager_addService(f->asBinder().get(), "foo");
```

Client:

```cpp
auto binder = AServiceManager_getService("foo");
auto foo = IFoo::fromBinder(ndk::SpAIBinder(binder));
int32_t result;
foo->doSomething(10, &result); // result = 20
```

---

3. ⚠️ CPP Backend (Unstable, Internal Use)

- Used within Android platform C++ internals.
- Works like NDK but less safe for updates.
- Not recommended unless you're modifying core platform internals.

---

4. 🦀 Rust Backend (for System Code in Rust)

Implementation:

```rust
use my_package::aidl::my::package::IFoo::{BnFoo, IFoo};
use binder::Result;

pub struct FooImpl;

impl Interface for FooImpl {}

impl IFoo for FooImpl {
    fn doSomething(&self, x: i32) -> Result<i32> {
        Ok(x * 2)
    }
}
```

Register:

```rust
let service = BnFoo::new_binder(FooImpl, BinderFeatures::default());
binder::add_service("foo", service)?;
```

📦 Structured Parcelables

Instead of using Java **`Parcelable`** or writing C++/Rust serialization manually, AIDL supports **structured parcelables**:

```aidl
package my.package;

parcelable MyData {
    int id = 0;
    String name = "default";
}
```

- Works in all backends.
- Fields must be plain types (int, String, etc.).

🔀 Unions in AIDL (Android 12+)

```aidl
union Result {
    int successCode;
```

```
    String errorMessage;
}
```

- One active field at a time.
- Works like enums + variants in Rust or tagged structs in C++.

↔ `in`, `out`, `inout` in Methods

```aidl
void doWork(in MyData input, out MyData result);
```

- **`in`**: caller → service
- **`out`**: service → caller
- **`inout`**: both ways (avoid if possible)

❌ Custom Parcelables

Only use if you already have hand-written code in C++/Rust/Java. Instead, prefer **structured parcelables**.

🧵 Threading

You must start a threadpool:

Java:

```java
BinderInternal.setMaxThreads(5);
```

NDK:

```cpp
ABinderProcess_setThreadPoolMaxThreadCount(4);
ABinderProcess_startThreadPool();
```

Rust:

```rust
binder::ProcessState::start_thread_pool();
```

⚠️ Built-in Error Handling

Java:

- Throws **`RemoteException`** or **`ServiceSpecificException`**.

C++ (CPP):

- Returns **`android::binder::Status`**

C++ (NDK):

- Returns **`ndk::ScopedAStatus`**

Rust:

- Returns **`Result<T, StatusCode>`**

✅ Summary

Feature	Java	NDK (C++)	CPP	Rust
Best for	Apps	System / APEX	Android internals	System code in Rust
Type safety	Good	Great	Medium	Excellent
Thread-safe	Yes	Yes	Needs care	Yes
Stability	Stable	Stable	Unstable	Stable
Structured AIDL	✅	✅	✅	✅

Would you like a full working AIDL interface + client + service example in any of these backends (Java, NDK, or Rust)?

5. Stable AIDL

bookmark_border
Android 10 adds support for stable Android Interface Definition Language (AIDL), a new way to keep track of the application program interface (API) and application binary interface (ABI) provided by AIDL interfaces. Stable AIDL works exactly like AIDL, but the build system tracks interface compatibility, and there are restrictions on what you can do:

Interfaces are defined in the build system with aidl_interfaces.
Interfaces can contain only structured data. Parcelables representing the preferred types are automatically created based on their AIDL definition and are automatically marshalled and unmarshalled.
Interfaces can be declared as stable (backward compatible). When this happens, their API is tracked and versioned in a file next to the AIDL interface.
Structured versus stable AIDL
Structured AIDL refers to types defined purely in AIDL. For example, a parcelable declaration (a custom parcelable) isn't structured AIDL. Parcelables with their fields defined in AIDL are called structured parcelables.

Stable AIDL requires structured AIDL so that the build system and compiler can understand if changes made to parcelables are backward compatible. However, not all structured interfaces are stable. To be stable, an interface must use only structured types, and it also must use the following versioning features. Conversely, an interface isn't stable if the core build system is used to build it or if unstable:true is set.

Define an AIDL interface
A definition of aidl_interface looks like this:

```
aidl_interface {
    name: "my-aidl",
    srcs: ["srcs/aidl/**/*.aidl"],
    local_include_dir: "srcs/aidl",
    imports: ["other-aidl"],
    versions_with_info: [
        {
            version: "1",
            imports: ["other-aidl-V1"],
        },
        {
            version: "2",
            imports: ["other-aidl-V3"],
        }
    ],
    stability: "vintf",
    backend: {
        java: {
            enabled: true,
            platform_apis: true,
        },
        cpp: {
            enabled: true,
        },
        ndk: {
            enabled: true,
        },
        rust: {
            enabled: true,
        },
    },

}
```
name: The name of the AIDL interface module that uniquely identifies an AIDL interface.
srcs: The list of AIDL source files that compose the interface. The path for an AIDL type Foo defined in a package com.acme should be at <base_path>/com/acme/Foo.aidl, where <base_path> could be any directory related to the directory where Android.bp is. In the preceding example, <base_path> is srcs/aidl.
local_include_dir: The path from where the package name starts. It corresponds to <base_path> explained above.
imports: A list of aidl_interface modules that this uses. If one of your AIDL interfaces uses an interface or a parcelable from another aidl_interface,

put its name here. This can be the name by itself, to refer to the latest version, or the name with the version suffix (such as -V1) to refer to a specific version. Specifying a version has been supported since Android 12
versions: The previous versions of the interface that are frozen under api_dir, Starting in Android 11, the versions are frozen under aidl_api/name. If there are no frozen versions of an interface, this shouldn't be specified, and there won't be compatibility checks. This field has been replaced with versions_with_info for Android 13 and higher.
versions_with_info: List of tuples, each of which contains the name of a frozen version and a list with version imports of other aidl_interface modules that this version of the aidl_interface imported. The definition of the version V of an AIDL interface IFACE is located at aidl_api/IFACE/V. This field was introduced in Android 13, and it isn't supposed to be modified in Android.bp directly. The field is added or updated by invoking *-update-api or *-freeze-api. Also, versions fields is automatically migrated to versions_with_info when a user invokes *-update-api or *-freeze-api.
stability: The optional flag for the stability promise of this interface. This only supports "vintf". If stability is unset, the build system checks that the interface is backward compatible unless unstable is specified. Being unset corresponds to an interface with stability within this compilation context (so either all system things, for example, things in system.img and related partitions, or all vendor things, for example, things in vendor.img and related partitions). If stability is set to "vintf", this corresponds to a stability promise: the interface must be kept stable as long as it is used.
gen_trace: The optional flag to turn the tracing on or off. Starting in Android 14 the default is true for the cpp and java backends.
host_supported: The optional flag that when set to true makes the generated libraries available to the host environment.
unstable: The optional flag used to mark that this interface doesn't need to be stable. When this is set to true, the build system neither creates the API dump for the interface nor requires it to be updated.
frozen: The optional flag that when set to true means that the interface has no changes since the previous version of the interface. This enables more build-time checks. When set to false this means the interface is in development and has new changes so running foo-freeze-api generates a new version and automatically change the value to true. Introduced in Android 14.
backend.<type>.enabled: These flags toggle the each of the backends that the AIDL compiler generates code for. Four backends are supported: Java, C++, NDK, and Rust. Java, C++, and NDK backends are enabled by default. If any of these three backends isn't needed, it needs to be disabled explicitly. Rust is disabled by default until Android 15.
backend.<type>.apex_available: The list of APEX names that the generated stub library is available for.
backend.[cpp|java].gen_log: The optional flag that controls whether to generate additional code for gathering information about the transaction.
backend.[cpp|java].vndk.enabled: The optional flag to make this interface a part of VNDK. Default is false.
backend.[cpp|ndk].additional_shared_libraries: Introduced in Android 14, this flag adds dependencies to the native libraries. This flag is useful with ndk_header and cpp_header.
backend.java.sdk_version: The optional flag for specifying the version of the SDK that the Java stub library is built against. The default is "system_current". This shouldn't be set when backend.java.platform_apis is true.
backend.java.platform_apis: The optional flag that should be set to true when the generated libraries need to build against the platform API rather than the SDK.
For each combination of the versions and the enabled backends, a stub library is created. For how to refer to the specific version of the stub library for a specific backend, see Module naming rules.

Write AIDL files
Interfaces in stable AIDL are similar to traditional interfaces, with the exception that they aren't allowed to use unstructured parcelables (because these aren't stable! see Structured versus stable AIDL). The primary difference in stable AIDL is how parcelables are defined. Previously, parcelables were forward declared; in stable (and therefore structured) AIDL, parcelables fields and variables are defined explicitly.

```
// in a file like 'some/package/Thing.aidl'
package some.package;

parcelable SubThing {
    String a = "foo";
    int b;
}
```

A default is supported (but not required) for boolean, char, float, double, byte, int, long, and String. In Android 12, defaults for user-defined enumerations are also supported. When a default is not specified, a 0-like or empty value is used. Enumerations without a default value are initialized to 0 even if there is no zero enumerator.

Use stub libraries
After adding stub libraries as a dependency to your module, you can include them into your files. Here are examples of stub libraries in the build system (Android.mk can also be used for legacy module definitions). Note, in these examples, the version is not present, so it represents using an unstable interface, but names for interfaces with versions include additional information, see Versioning interfaces.

```
cc_... {
    name: ...,
    // use shared_libs: to load your library and its transitive dependencies
    // dynamically
    shared_libs: ["my-module-name-cpp"],
    // use static_libs: to include the library in this binary and drop
    // transitive dependencies
```

```
    static_libs: ["my-module-name-cpp"],
    ...
}
# or
java_... {
    name: ...,
    // use static_libs: to add all jars and classes to this jar
    static_libs: ["my-module-name-java"],
    // use libs: to make these classes available during build time, but
    // not add them to the jar, in case the classes are already present on the
    // boot classpath (such as if it's in framework.jar) or another jar.
    libs: ["my-module-name-java"],
    // use srcs: with -java-sources if you want to add classes in this
    // library jar directly, but you get transitive dependencies from
    // somewhere else, such as the boot classpath or another jar.
    srcs: ["my-module-name-java-source", ...],
    ...
}
# or
rust_... {
    name: ...,
    rustlibs: ["my-module-name-rust"],
    ...
}
```

Example in C++:

```
#include "some/package/IFoo.h"
#include "some/package/Thing.h"
...
    // use just like traditional AIDL
```

Example in Java:

```
import some.package.IFoo;
import some.package.Thing;
...
    // use just like traditional AIDL
```

Example in Rust:

```
use aidl_interface_name::aidl::some::package::{IFoo, Thing};
...
    // use just like traditional AIDL
```

Versioning interfaces

Declaring a module with name foo also creates a target in the build system that you can use to manage the API of the module. When built, foo-freeze-api adds a new API definition under api_dir or aidl_api/name, depending on the Android version, and adds a .hash file, both representing the newly frozen version of the interface. foo-freeze-api also updates the versions_with_info property to reflect the additional version and imports for the version. Basically, imports in versions_with_info is copied from imports field. But the latest stable version is specified in imports in versions_with_info for the import, which doesn't have an explicit version. After the versions_with_info property is specified, the build system runs compatibility checks between frozen versions and also between Top of Tree (ToT) and the latest frozen version.

In addition, you need to manage ToT version's API definition. Whenever an API is updated, run foo-update-api to update aidl_api/name/current which contains ToT version's API definition.

To maintain the stability of an interface, owners can add new:

Methods to the end of an interface (or methods with explicitly defined new serials)
Elements to the end of a parcelable (requires a default to be added for each element)
Constant values
In Android 11, enumerators
In Android 12, fields to the end of a union
No other actions are permitted, and no one else can modify an interface (otherwise they risk collision with changes that an owner makes).

To test that all interfaces are frozen for release, you can build with the following environmental variables set:

AIDL_FROZEN_REL=true m ... - build requires all stable AIDL interfaces to be frozen which have no owner: field specified.
AIDL_FROZEN_OWNERS="aosp test" - build requires all stable AIDL interfaces to be frozen with the owner: field specified as "aosp" or "test".
Stability of imports
Updating the versions of imports for frozen versions of an interface is backward compatible at the Stable AIDL layer. However, updating these requires updating all servers and clients which use a previous version of the interface, and some apps may be confused when mixing different versions of types. Generally, for types-only or common packages, this is safe because code needs to already be written to handle unknown types from IPC transactions.

In Android platform code android.hardware.graphics.common is the biggest example of this type of version upgrade.

Use versioned interfaces
Interface methods
At runtime, when trying to call new methods on an old server, new clients get either an error or an exception, depending on the backend.

cpp backend gets ::android::UNKNOWN_TRANSACTION.
ndk backend gets STATUS_UNKNOWN_TRANSACTION.
java backend gets android.os.RemoteException with a message saying the API is not implemented.
For strategies to handle this see querying versions and using defaults.

Parcelables
When new fields are added to parcelables, old clients and servers drop them. When new clients and servers receive old parcelables, the default values for new fields are automatically filled in. This means that defaults need to be specified for all new fields in a parcelable.

Clients shouldn't expect servers to use the new fields unless they know the server is implementing the version that has the field defined (see querying versions).

Enums and constants
Similarly, clients and servers should either reject or ignore unrecognized constant values and enumerators as appropriate, since more may be added in the future. For example, a server shouldn't abort when it receives an enumerator that it doesn't know about. The server should either ignore the enumerator, or return something so the client knows it's unsupported in this implementation.

Unions
Trying to send a union with a new field fails if the receiver is old and doesn't know about the field. The implementation will never see the union with the new field. The failure is ignored if it's a one-way transaction; otherwise the error is BAD_VALUE(for the C++ or NDK backend) or IllegalArgumentException(for the Java backend). The error is received if the client is sending a union set to the new field to an old server, or when it's an old client receiving the union from a new server.

Manage multiple versions
A linker namespace in Android can have only 1 version of a specific aidl interface to avoid situations where the generated aidl types have multiple definitions. C++ has the One Definition Rule that requires only one definition of each symbol.

The Android build provides an error when a module is depending on different versions of the same aidl_interface library. The module might be depending on these libraries directly or indirectly through dependencies of their dependencies. These errors show the dependency graph from the failing module to the conflicting versions of the aidl_interface library. All of the dependencies need to be updated to include the same (usually the latest) version of these libraries.

If the interface library is used by many different modules, it can be helpful to create cc_defaults, java_defaults, and rust_defaults for any group of libraries and processes that need to use the same version. When introducing a new version of the interface those defaults can be updated and all of modules using them are updated together, ensuring they aren't using different versions of the interface.

```
cc_defaults {
  name: "my.aidl.my-process-group-ndk-shared",
  shared_libs: ["my.aidl-V3-ndk"],
  ...
}

cc_library {
  name: "foo",
  defaults: ["my.aidl.my-process-group-ndk-shared"],
  ...
}

cc_binary {
  name: "bar",
  defaults: ["my.aidl.my-process-group-ndk-shared"],
```

```
  ...
}
```

When aidl_interface modules import other aidl_interface modules this creates additional dependencies that require specific versions to be used together. This situation can become difficult to manage when there are common aidl_interface modules that are imported in multiple aidl_interface modules that are used together in the same processes.

aidl_interfaces_defaults can be used to keep one definition of the latest versions of dependencies for an aidl_interface that can be updated in a single place, and used by all aidl_interface modules that want to import that common interface.

```
aidl_interface_defaults {
  name: "android.popular.common-latest-defaults",
  imports: ["android.popular.common-V3"],
  ...
}

aidl_interface {
  name: "android.foo",
  defaults: ["my.aidl.latest-ndk-shared"],
  ...
}

aidl_interface {
  name: "android.bar",
  defaults: ["my.aidl.latest-ndk-shared"],
  ...
}
```

Flag-based development
In-development (unfrozen) interfaces can't be used on release devices, because they aren't guaranteed to be backward compatible.

AIDL supports run time fallback for these unfrozen interface libraries in order for code to be written against the latest unfrozen version and still be used on release devices. The backward-compatible behavior of clients is similar to existing behavior and with the fallback, the implementations also need to follow those behaviors. See Use versioned interfaces.

AIDL build flag
The flag that controls this behavior is RELEASE_AIDL_USE_UNFROZEN defined in build/release/build_flags.bzl. true means the unfrozen version of the interface is used at run time and false means the libraries of the unfrozen versions all behave like their last frozen version. You can override the flag to true for local development, but must revert it to false before release. Typically development is done with a configuration that has the flag set to true.

Compatibility matrix and manifests
Vendor interface objects (VINTF objects) define what versions are expected, and what versions are provided on either side of the vendor interface.

Most non-Cuttlefish devices target the latest compatibility matrix only after interfaces are frozen, so there is no difference in the AIDL libraries based on RELEASE_AIDL_USE_UNFROZEN.

Matrixes
Partner-owned interfaces are added to device-specific or product-specific compatibility matrixes that the device targets during development. So when a new, unfrozen version of an interface is added to a compatibility matrix, the previous frozen versions needs to remain for RELEASE_AIDL_USE_UNFROZEN=false. You can handle this by using different compatibility matrix files for different RELEASE_AIDL_USE_UNFROZEN configurations or allowing both versions in a single compatibility matrix file that is used in all configurations.

For example, when adding an unfrozen version 4, use <version>3-4</version>.

When version 4 is frozen you can remove version 3 from the compatibility matrix because the frozen version 4 is used when RELEASE_AIDL_USE_UNFROZEN is false.

Manifests
In Android 15, a change in libvintf is introduced to modify the manifest files at build time based on the value of RELEASE_AIDL_USE_UNFROZEN.

The manifests and the manifest fragments declare which version of an interface a service implements. When using the latest unfrozen version of an interface, the manifest must be updated to reflect this new version. When RELEASE_AIDL_USE_UNFROZEN=false the manifest entries are adjusted by libvintf to reflect the change in the generated AIDL library. The version is modified from the unfrozen version, N, to the last frozen version N - 1. Therefore, users don't need to manage multiple manifests or manifest fragments for each of their services.

HAL client changes

HAL client code must be backward compatible with each previous supported frozen version. When RELEASE_AIDL_USE_UNFROZEN is false the services always look like the last frozen version or earlier (for example, calling new unfrozen methods returns UNKNOWN_TRANSACTION, or new parcelable fields have their default values). Android framework clients are required to be backward compatible with additional previous versions, but this is a new detail for vendor clients and clients of partner-owned interfaces.

HAL implementation changes
The largest difference in HAL development with flag-based development is the requirement for HAL implementations to be backward compatible with the last frozen version to work when RELEASE_AIDL_USE_UNFROZEN is false. Considering backward compatibility in implementations and device code is a new exercise. See Use versioned interfaces.

The backward compatibility considerations are generally the same for the clients and servers, and for framework code and vendor code, but there are subtle differences that you need to be aware of, as you're now effectively implementing two versions that use the same source code (the current, unfrozen version).

Example: An interface has three frozen versions. The interface is updated with a new method. The client and service are both updated to use the new version 4 library. Because the V4 library is based on an unfrozen version of the interface, it behaves like the last frozen version, version 3, when RELEASE_AIDL_USE_UNFROZEN is false, and prevents the use of the new method.

When the interface is frozen, all values of RELEASE_AIDL_USE_UNFROZEN use that frozen version, and the code handling the backward compatibility can be removed.

When calling methods on callbacks, you must gracefully handle the case when UNKNOWN_TRANSACTION is returned. Clients might be implementing two different versions of a callback based on the release configuration, so you can't assume the client sends the newest version, and new methods might return this. This is similar to how stable AIDL clients maintain backward compatibility with servers describes in Use versioned interfaces.

```
// Get the callback along with the version of the callback
ScopedAStatus RegisterMyCallback(const std::shared_ptr<IMyCallback>& cb) override {
    mMyCallback = cb;
    // Get the version of the callback for later when we call methods on it
    auto status = mMyCallback->getInterfaceVersion(&mMyCallbackVersion);
    return status;
}

// Example of using the callback later
void NotifyCallbackLater() {
  // From the latest frozen version (V2)
  mMyCallback->foo();
  // Call this method from the unfrozen V3 only if the callback is at least V3
  if (mMyCallbackVersion >= 3) {
    mMyCallback->bar();
  }
}
```

New fields in existing types (parcelable, enum, union) might not exist or contain their default values when RELEASE_AIDL_USE_UNFROZEN is false and the values of new fields a service tries to send are dropped on the way out of the process.

New types added in this unfrozen version can't be sent or received through the interface.

The implementation never gets a call for new methods from any clients when RELEASE_AIDL_USE_UNFROZEN is false.

Be careful to use new enumerators only with the version they're introduced in, and not the previous version.

Normally, you use foo->getInterfaceVersion() to see which version the remote interface is using. However with flag-based versioning support, you're implementing two different versions, so you might want to get the version of the current interface. You can do this by getting the interface version of the current object, for example, this->getInterfaceVersion() or the other methods for my_ver. See Querying the interface version of the remote object for more information.

New VINTF stable interfaces
When a new AIDL interface package is added there is no last frozen version, so there is no behavior to fall back to when RELEASE_AIDL_USE_UNFROZEN is false. Don't use these interfaces. When RELEASE_AIDL_USE_UNFROZEN is false, Service Manager won't allow the service to register the interface and the clients won't find it.

You can add the services conditionally based on the value of the RELEASE_AIDL_USE_UNFROZEN flag in the device makefile:

```
ifeq ($(RELEASE_AIDL_USE_UNFROZEN),true)
PRODUCT_PACKAGES += \
   android.hardware.health.storage-service
endif
```

If the service is a part of a larger process so you can't add it to the device conditionally, you can check to see if the service is declared with IServiceManager::isDeclared(). If it's declared and failed to register, then abort the process. If it isn't declared, then it's expected to fail to register.

New VINTF stable extension interfaces

New extension interfaces have no previous version to fall back to and because they aren't registered with ServiceManager or declared in VINTF manifests, IServiceManager::isDeclared() can't be used to determine when to attach the extension interface to another interface.

The RELEASE_AIDL_USE_UNFROZEN variable can be used to determine whether to attach the new unfrozen extension interface to the existing interface to avoid using it on released devices. The interface needs to be frozen to be used on released devices.

The vts_treble_vintf_vendor_test and vts_treble_vintf_framework_test VTS tests detect when an unfrozen extension interface is used in a released device and throw an error.

If the extension interface isn't new and has a previously frozen version, then it falls back to that previously frozen version and there are no extra steps required.

Cuttlefish as a development tool

Every year after the VINTF is frozen we adjust the framework compatibility matrix (FCM) target-level and the PRODUCT_SHIPPING_API_LEVEL of Cuttlefish so they reflect devices launching with next year's release. We adjust target-level and PRODUCT_SHIPPING_API_LEVEL to make sure there is some launching device that is tested and meets the new requirements for next year's release.

When RELEASE_AIDL_USE_UNFROZEN is true, Cuttlefish is used for development of future Android releases. It targets next year's Android release's FCM level and PRODUCT_SHIPPING_API_LEVEL, requiring it to satisfy the next release's Vendor Software Requirements (VSR).

When RELEASE_AIDL_USE_UNFROZEN is false, Cuttlefish has the previous target-level and PRODUCT_SHIPPING_API_LEVEL to reflect a release device. In Android 14 and lower, this differentiation would be accomplished with different Git branches that don't pick up the change to FCM target-level, shipping API level, or any other code targeting the next release.

Module naming rules

In Android 11, for each combination of the versions and the backends enabled, a stub library module is automatically created. To refer to a specific stub library module for linking, don't use the name of the aidl_interface module, but the name of the stub library module, which is ifacename-version-backend, where

ifacename: name of the aidl_interface module
version is either of
Vversion-number for the frozen versions
Vlatest-frozen-version-number + 1 for the tip-of-tree (yet-to-be-frozen) version
backend is either of
java for the Java backend,
cpp for the C++ backend,
ndk or ndk_platform for the NDK backend. The former is for apps, and the latter is for platform usage until Android 13. In Android 13 and later, only use ndk.
rust for Rust backend.
Assume that there is a module with name foo and its latest version is 2, and it supports both NDK and C++. In this case, AIDL generates these modules:

Based on version 1
foo-V1-(java|cpp|ndk|ndk_platform|rust)
Based on version 2 (the latest stable version)
foo-V2-(java|cpp|ndk|ndk_platform|rust)
Based on ToT version
foo-V3-(java|cpp|ndk|ndk_platform|rust)
Compared to Android 11:

foo-backend, which referred to the latest stable version becomes foo-V2-backend
foo-unstable-backend, which referred to the ToT version becomes foo-V3-backend
The output file names are always the same as module names.

Based on version 1: foo-V1-(cpp|ndk|ndk_platform|rust).so
Based on version 2: foo-V2-(cpp|ndk|ndk_platform|rust).so
Based on ToT version: foo-V3-(cpp|ndk|ndk_platform|rust).so
Note that the AIDL compiler doesn't create either an unstable version module, or a non-versioned module for a stable AIDL interface. As of

Android 12, the module name generated from a stable AIDL interface always includes its version.

New meta interface methods
Android 10 adds several meta interface methods for the stable AIDL.

Query the interface version of the remote object
Clients can query the version and hash of the interface that the remote object is implementing and compare the returned values with the values of the interface that the client is using.

Example with the cpp backend:

```
sp<IFoo> foo = ... // the remote object
int32_t my_ver = IFoo::VERSION;
int32_t remote_ver = foo->getInterfaceVersion();
if (remote_ver < my_ver) {
  // the remote side is using an older interface
}

std::string my_hash = IFoo::HASH;
std::string remote_hash = foo->getInterfaceHash();
```

Example with the ndk (and the ndk_platform) backend:

```
IFoo* foo = ... // the remote object
int32_t my_ver = IFoo::version;
int32_t remote_ver = 0;
if (foo->getInterfaceVersion(&remote_ver).isOk() && remote_ver < my_ver) {
  // the remote side is using an older interface
}

std::string my_hash = IFoo::hash;
std::string remote_hash;
foo->getInterfaceHash(&remote_hash);
```

Example with the java backend:

```
IFoo foo = ... // the remote object
int myVer = IFoo.VERSION;
int remoteVer = foo.getInterfaceVersion();
if (remoteVer < myVer) {
  // the remote side is using an older interface
}

String myHash = IFoo.HASH;
String remoteHash = foo.getInterfaceHash();
```

For Java language, the remote side MUST implement getInterfaceVersion() and getInterfaceHash() as follows (super is used instead of IFoo to avoid copy and paste mistakes. The annotation @SuppressWarnings("static") might be needed to disable warnings, depending on the javac configuration):

```
class MyFoo extends IFoo.Stub {
    @Override
    public final int getInterfaceVersion() { return super.VERSION; }

    @Override
    public final String getInterfaceHash() { return super.HASH; }
}
```

This is because the generated classes (IFoo, IFoo.Stub, etc.) are shared between the client and server (for example, the classes can be in the boot classpath). When classes are shared, the server is also linked against the newest version of the classes even though it might have been built with an older version of the interface. If this meta interface is implemented in the shared class, it always returns the newest version. However, by implementing the method as above, the version number of the interface is embedded in the server's code (because IFoo.VERSION is a static final int that is inlined when referenced) and thus the method can return the exact version the server was built with.

Deal with older interfaces
It's possible that a client is updated with the newer version of an AIDL interface but the server is using the old AIDL interface. In such cases,

calling a method on an old interface returns UNKNOWN_TRANSACTION.

With stable AIDL, clients have more control. In the client side, you can set a default implementation to an AIDL interface. A method in the default implementation is invoked only when the method isn't implemented in the remote side (because it was built with an older version of the interface). Since defaults are set globally, they should not be used from potentially shared contexts.

Warning: setDefaultImpl is deprecated. When using this method, it's impossible to see which object you're implementing the default for. Instead, use I*Delegator which is generated for various backends. Note, in Java I*Delegator needs to be enabled with @JavaDelegator on the interface.
Example in C++ in Android 13 and later:

```
class MyDefault : public IFooDefault {
  Status anAddedMethod(...) {
   // do something default
  }
};

// once per an interface in a process
IFoo::setDefaultImpl(::android::sp<MyDefault>::make());

foo->anAddedMethod(...); // MyDefault::anAddedMethod() will be called if the
                         // remote side is not implementing it
```

Note: Before Android 13 IFoo::setDefaultImpl received a std::unique_ptr<IFoo> instead of ::android::sp<IFoo>.
Example in Java:

```
IFoo.Stub.setDefaultImpl(new IFoo.Default() {
    @Override
    public xxx anAddedMethod(...)  throws RemoteException {
        // do something default
    }
}); // once per an interface in a process

foo.anAddedMethod(...);
```

Note: In Java, setDefaultImpl is in the Stub class, and not in the interface class.
You don't need to provide the default implementation of all methods in an AIDL interface. Methods that are guaranteed to be implemented in the remote side (because you're certain that the remote is built when the methods were in the AIDL interface description) don't need to be overridden in the default impl class.

Note: The default implementation is not used for oneway method calls. This is because there is no way for a client to know if the remote has the method and has executed it or not; there is no acknowledgement from the remote side in case of the oneway method calls.
Convert existing AIDL to structured or stable AIDL
If you have an existing AIDL interface and code that uses it, use the following steps to convert the interface to a stable AIDL interface.

Identify all of the dependencies of your interface. For every package the interface depends on, determine if the package is defined in stable AIDL. If not defined, the package must be converted.

Convert all parcelables in your interface into stable parcelables (the interface files themselves can remain unchanged). Do this by expressing their structure directly in AIDL files. Management classes must be rewritten to use these new types. This can be done before you create an aidl_interface package (below).

Create an aidl_interface package (as described above) that contains the name of your module, its dependencies, and any other information you need. To make it stabilized (not just structured), it also needs to be versioned. For more information, see Versioning interfaces.

Great — you're diving deep into **Stable AIDL** and the Soong-based build system for structured, versioned IPC in Android. Let's **summarize and clarify all the key concepts** to help you confidently implement, manage, and version a stable AIDL interface.

✅ Stable AIDL: Concept Summary

🧩 1. What Is Stable AIDL?

Stable AIDL:

- Enables **API and ABI stability** across Android releases.
- Uses **structured types only** (no `Parcelable` classes from Java).
- Supports **versioning**, **backward compatibility**, and **VINTF integration**.
- Used primarily for system services and HALs that interact between system and vendor partitions.

🏗️ 2. How To Declare a `stable AIDL` Interface in `Android.bp`

```bp
aidl_interface {
    name: "my.foo",
    srcs: ["srcs/aidl/**/*.aidl"],
    local_include_dir: "srcs/aidl",
    imports: ["android.hardware.foo"],
    stability: "vintf",   // Required for VINTF stable interfaces
    backend: {
        java: { enabled: true, platform_apis: true },
        cpp: { enabled: true },
        ndk: { enabled: true },
        rust: { enabled: false }, // enable for Rust usage
    },
}
```

After this, run:

- `m my.foo` to build
- `m my.foo-update-api` to generate a version
- `m my.foo-freeze-api` to finalize a version (`aidl_api/my.foo/V1`)

🔐 3. Structured vs Stable AIDL

Type	Description	Used in Stable AIDL?
`structured parcelable`	Defined directly in `.aidl`, field-by-field	✅ Yes
`custom parcelable`	Written in Java/C++ with `readFromParcel()`	❌ No

✅ Example Structured Parcelable

```aidl
package my.foo;

parcelable Bar {
    int id = 0;
    String name = "default";
}
```

🧬 4. Versioning (`V1`, `V2`, `V3`...)

Each time you change the interface:

- Add methods at the **end**
- Add fields to parcelables with **defaults**
- Run:
  - `m my.foo-update-api` — updates `aidl_api/my.foo/current`
  - `m my.foo-freeze-api` — creates `aidl_api/my.foo/VN`, bumps `versions_with_info`

🧩 5. Interface Compatibility Rules

Action	Allowed in Stable AIDL?
Add method to interface	✅ Yes (at end)
Remove method	❌ No

Action	Allowed in Stable AIDL?
Add field to parcelable	✅ Yes (must have default)
Change type of existing field	❌ No
Reorder fields/methods	❌ No

🏷️ 6. Module Naming Rules

Version	NDK Stub	Java Stub	C++ Stub
`V1`	`foo-V1-ndk`	`foo-V1-java`	`foo-V1-cpp`
`V2`	`foo-V2-ndk`	`foo-V2-java`	`foo-V2-cpp`
`ToT` (dev)	`foo-V3-ndk`	`foo-V3-java`	`foo-V3-cpp`

These can be used in `shared_libs`, `static_libs`, `libs`, or `rustlibs` in other modules.

📦 7. Runtime Version Queries (Cross-version Safety)

You can safely check if a remote service supports a method using:

Java:

```java
int remoteVer = foo.getInterfaceVersion();
if (remoteVer >= 2) foo.newMethod();
```

NDK:

```cpp
int32_t ver;
foo->getInterfaceVersion(&ver);
if (ver >= 2) foo->newMethod(...);
```

🔄 8. Fallback & Compatibility (RELEASE_AIDL_USE_UNFROZEN)

✅ Dev Flag: `RELEASE_AIDL_USE_UNFROZEN`

- `true`: use in-development AIDL
- `false`: fallback to last frozen version only

  Clients must:

- Expect missing methods or dropped fields
- Handle `UNKNOWN_TRANSACTION`, default values

⚠️ 9. Fallback Example

```cpp
ScopedAStatus RegisterCallback(const shared_ptr<IMyCallback>& cb) {
    mCb = cb;
    cb->getInterfaceVersion(&mCbVersion);  // cache version
    return ScopedAStatus::ok();
}

void NotifyLater() {
    mCb->foo();  // from frozen V2
    if (mCbVersion >= 3)
        mCb->bar();  // from unfrozen V3
}
```

🧪 10. Freeze Policy for Releases

Enforce freezing for all stable AIDL interfaces:

```bash
AIDL_FROZEN_REL=true m
```

Or set owners:

```bash
AIDL_FROZEN_OWNERS="aosp yourteam"
```

🔁 11. Convert Existing AIDL to Stable

1. Convert custom parcelables → structured AIDL.
2. Add `aidl_interface` module.
3. Set `stability: "vintf"` if using in HALs.
4. Version & freeze using `*-update-api` and `*-freeze-api`.

🛠️ 12. Fallback Registration Control in `Android.mk` / `.bp`

```make
ifeq ($(RELEASE_AIDL_USE_UNFROZEN),true)
PRODUCT_PACKAGES += android.hardware.foo-service
endif
```

📌 Conclusion

Stable AIDL enforces structure, compatibility, and versioning across Android releases. Key actions:

- Always use **structured parcelables**
- Use `aidl_interface` to declare
- Run `*-freeze-api` before release
- Use `getInterfaceVersion()` to handle backward compatibility
- Avoid unstructured data or Java parcelable implementations

Would you like me to:

- Convert your existing AIDL to a stable `aidl_interface` module?
- Create a full `Android.bp` + `.aidl` + versioning flow for C++/NDK?
- Or generate client + server implementation templates?

Let me know how you'd like to proceed.

6. ↪
Freeze
explain here in simple words

Sure! Let's focus on **"freeze" in Stable AIDL** and explain it in **simple words**.

🔒 What Does "Freeze" Mean in AIDL?

When you **freeze** an AIDL interface, you're saying:

> "This version is ready to be used in real Android devices. I won't change it anymore."

It's like **locking a version** of your API — so others can depend on it, and the system can check that future versions are **backward compatible**.

🧊 Why Do We Freeze?

- Android needs to guarantee that **vendor and system code** can talk to each other across updates.
- If you don't freeze an interface, it can **keep changing**, which **breaks devices** when Android gets updated.

🛠️ How Do You Freeze?

You freeze your interface with a simple build command:

```bash
m my.aidl-interface-freeze-api
```

This:

- Creates a **frozen version folder** like: **`aidl_api/my.aidl-interface/V1`**
- Adds a **`.hash`** file (used to check compatibility later)
- Updates **`versions_with_info`** in your **`Android.bp`**

🧪 What Happens After Freezing?

After you freeze:

- Android build system will **check for compatibility** if you make changes.
- You **can't break old clients** — only **add safe changes**.

✅ What Changes Are Allowed *After* Freezing?

You can still:

- ✅ Add a new method (at the end)
- ✅ Add a new field in a **`parcelable`** (must set a default)
- ✅ Add a new enum value
- ✅ Add a new union field (with default in Android 12+)

You cannot:

- ❌ Rename or remove methods or fields
- ❌ Change existing field types
- ❌ Reorder fields/methods

📦 How It Helps?

- Ensures **Android updates don't break vendor drivers**
- Enables **VINTF compatibility checks**
- Keeps **interfaces clean and stable**

🔁 What if You're Still Changing Things?

If your AIDL is **not frozen yet**, it is **"unfrozen" or development version**.

- You can change anything.
- But you can't ship this on real devices.

Use the flag `RELEASE_AIDL_USE_UNFROZEN=true` in development to allow testing unfrozen AIDL.

📌 Summary (Very Simple)

Term	Meaning
Freeze	Lock current version of your interface. No breaking changes allowed.
Why	So Android can safely evolve without breaking vendor or app code.
How	Use `m my.aidl-interface-freeze-api` to freeze.
Then	System tracks changes and warns if you break compatibility.

Would you like an example command flow or a sample project with freeze setup?

7. ↪
* `m my.foo-update-api` — updates `aidl_api/my.foo/current` * `m my.foo-freeze-api` — creates `aidl_api/my.foo/VN`, bumps `versions_with_info` explain

Great — let's break it down **simply**.

❓ What are `my.foo-update-api` and `my.foo-freeze-api`?

These are **special build targets** used to manage the **versioning** of your **stable AIDL interface**.

🆕 1. `m my.foo-update-api`

🔄 **Updates the current development version** of your AIDL interface.

- Saves the latest state of your AIDL to:
  👉 `aidl_api/my.foo/current/`

Think of this like:

> "I've made changes. Save the latest version as the current development version."

📝 Useful when:

- You've added new methods or fields.
- You want to **keep working**, but still want the build system to **track changes**.

🧊 2. `m my.foo-freeze-api`

❄️ **Freezes** the current development version into a new **official version**.

- It moves `current/` → a new version like `V1`, `V2`, etc.
  👉 `aidl_api/my.foo/V1/`
- Updates `versions_with_info` in `Android.bp`

Think of this like:

> "This version is now stable. Lock it. Don't allow breaking changes anymore."

After this, future changes must be **backward-compatible** only.

🔄 Full Versioning Flow

Let's say your AIDL module is called `my.foo`. Here's how it goes:

💻 You make changes to the AIDL file.

```aidl
interface IFoo {
    void doSomething();
}
```

🔄 You run:

```bash
m my.foo-update-api
```

📂 This updates:

```bash
aidl_api/my.foo/current/IFoo.aidl
```

✅ **You're happy. You run:**

```bash
m my.foo-freeze-api
```

📂 This does:

```bash
aidl_api/my.foo/V1/IFoo.aidl
```

📄 Updates:

```bp
versions_with_info: [
  {
    version: "1",
    ...
  },
]
```

Now you have a **frozen version (V1)**.

📌 Summary Table

Command	What it does
`m my.foo-update-api`	Updates `current/` AIDL version (for dev tracking)
`m my.foo-freeze-api`	Freezes current as `V1`, `V2` etc. + updates metadata

📦 After Freezing, You Get:

Path	Purpose
`aidl_api/my.foo/V1/`	Frozen API version 1
`aidl_api/my.foo/current/`	Ongoing development version (ToT)
`Android.bp` → `versions_with_info`	Tracks frozen version history

Let me know if you'd like a visual diagram of the versioning folder structure or a sample `Android.bp` showing this in context!

8. AIDL for HALs

bookmark_border
Android 11 introduces the ability to use AIDL for HALs in Android, making it possible to implement parts of Android without HIDL. Transition HALs to use AIDL exclusively where possible (when upstream HALs use HIDL, HIDL must be used).

HALs using AIDL to communicate between framework components, such as those in system.img, and hardware components, such as those in vendor.img, must use stable AIDL. However, to communicate within a partition, for example, from one HAL to another, there's no restriction on the IPC mechanism to use.

Motivation
AIDL has been around longer than HIDL, and is used in many other places, such as between Android framework components or in apps. Now that AIDL has stability support, it's possible to implement an entire stack with a single IPC runtime. AIDL also has a better versioning system than HIDL. Here are some advantages of AIDL:

Using a single IPC language means having only one thing to learn, debug, optimize, and secure.
AIDL supports in-place versioning for the owners of an interface:
Owners can add methods to the end of interfaces, or fields to parcelables. This means it's easier to version code over the years, and also the year over year cost is smaller (types can be amended in place and there's no need for extra libraries for each interface version).
Extension interfaces can be attached at run time rather than in the type system, so there's no need to rebase downstream extensions onto newer versions of interfaces.
An existing AIDL interface can be used directly when its owner chooses to stabilize it. Before, an entire copy of the interface would have to be created in HIDL.
Build against the AIDL runtime
Warning: Only system/system_ext can use libbinder directly. Other partitions must use libbinder_ndk, libbinder_rs, or the Java backend to communicate stably.
Warning: The Java backend can't be used to communicate with the service manager on the vendor partition, and no new system APIs can be used on the vendor partition after API level 34.
AIDL has three different backends: Java, NDK, and CPP. To use stable AIDL, always use the system copy of libbinder at system/lib*/libbinder.so and talk on /dev/binder. For code on the vendor image, this means that libbinder (from the VNDK) can't be used: this library has an unstable C++ API and unstable internals. Instead, native vendor code must use the NDK backend of AIDL, link against libbinder_ndk (which is backed by system libbinder.so), and link against the NDK libraries created by aidl_interface entries. For the exact module names, see Module naming rules.

Write an AIDL HAL interface
For an AIDL interface to be used between system and vendor, the interface needs two changes:

Every type definition must be annotated with @VintfStability.
The aidl_interface declaration needs to include stability: "vintf",.
Only the owner of an interface can make these changes.

When you make these changes, the interface must be in the VINTF manifest in order to work. Test this (and related requirements, such as verifying that released interfaces are frozen) using the VTS test vts_treble_vintf_vendor_test. You can use a @VintfStability interface without these requirements by calling either AIBinder_forceDowngradeToLocalStability in the NDK backend, android::Stability::forceDowngradeToLocalStability in the C++ backend, or android.os.Binder#forceDowngradeToSystemStability in the Java backend on a binder object before it's sent to another process.

Note: Downgrading a service to vendor stability isn't supported in Java because all apps run in a system context.
Additionally, for maximum code portability and to avoid potential problems such as unnecessary additional libraries, disable the CPP backend.

The code shows how to disable the CPP backend:

```
aidl_interface: {
    ...
    backend: {
        cpp: {
            enabled: false,
        },
    },
}
```
Find AIDL HAL interfaces
AOSP stable AIDL interfaces for HALS are within aidl folders in the same base directories as HIDL interfaces:

hardware/interfaces is for interfaces typically provided by hardware.
frameworks/hardware/interfaces is for high-level interfaces provided to hardware.
system/hardware/interfaces is for low-level interfaces provided to hardware.
Put extension interfaces into other hardware/interfaces subdirectories in vendor or hardware.

Extension interfaces
Android has a set of official AOSP interfaces with every release. When Android partners want to add capabilities to these interfaces, they shouldn't change these directly because this makes their Android runtime incompatible with the AOSP Android runtime. Avoid changing these interfaces so the GSI image can continue to work.

Extensions can register in two different ways:

At runtime; see Attached extension interfaces
As a standalone, registered globally and in VINTF
However an extension is registered, when vendor-specific (meaning not a part of upstream AOSP) components use the interface, merge conflicts aren't possible However, when downstream modifications to upstream AOSP components are made, merge conflicts can result, and the following strategies are recommended:

Upstream the interface additions to AOSP in the next release.
Upstream interface additions that allow further flexibility (without merge conflicts) in the next release.
Extension parcelables: ParcelableHolder
ParcelableHolder is an instance of the Parcelable interface that can contain another instance of Parcelable.

The main use case of ParcelableHolder is to make Parcelable extensible. For example, image that device implementers expect to be able to extend an AOSP-defined Parcelable, AospDefinedParcelable, to include their value-add features.

Use the ParcelableHolder interface to extend Parcelable with your value-add features. The ParcelableHolder interface contains an instance of Parcelable. If you try to add fields to Parcelable directly, it causes an error:

```
parcelable AospDefinedParcelable {
  int a;
  String b;
  String x; // ERROR: added by a device implementer
  int[] y; // added by a device implementer
}
```

As seen in the preceding code, this practice is broken because the fields added by the device implementer might have a conflict when Parcelable is revised in the next releases of Android.

Using ParcelableHolder, the owner of a parcelable can define an extension point in an instance of Parcelable:

```
parcelable AospDefinedParcelable {
  int a;
  String b;
  ParcelableHolder extension;
}
```

Then the device implementers can define their own Parcelable instance for their extension:

```
parcelable OemDefinedParcelable {
  String x;
  int[] y;
}
```

The new Parcelable instance can be attached to the original Parcelable with the ParcelableHolder field:

```
// Java
AospDefinedParcelable ap = ...;
OemDefinedParcelable op = new OemDefinedParcelable();
op.x = ...;
op.y = ...;

ap.extension.setParcelable(op);
```

```
...

OemDefinedParcelable op = ap.extension.getParcelable(OemDefinedParcelable.class);

// C++
AospDefinedParcelable ap;
OemDefinedParcelable op;
std::shared_ptr<OemDefinedParcelable> op_ptr = make_shared<OemDefinedParcelable>();

ap.extension.setParcelable(op);
ap.extension.setParcelable(op_ptr);

...

std::shared_ptr<OemDefinedParcelable> op_ptr;

ap.extension.getParcelable(&op_ptr);

// NDK
AospDefinedParcelable ap;
OemDefinedParcelable op;
ap.extension.setParcelable(op);

...

std::optional<OemDefinedParcelable> op;
ap.extension.getParcelable(&op);

// Rust
let mut ap = AospDefinedParcelable { .. };
let op = Rc::new(OemDefinedParcelable { .. });

ap.extension.set_parcelable(Rc::clone(&op));

...

let op = ap.extension.get_parcelable::<OemDefinedParcelable>();
```

AIDL HAL server instance names
By convention, AIDL HAL services have an instance name of the format $package.$type/$instance. For example, an instance of the vibrator HAL is registered as android.hardware.vibrator.IVibrator/default.

Write an AIDL HAL server
@VintfStability AIDL servers must be declared in the VINTF manifest, for example:

```
    <hal format="aidl">
        <name>android.hardware.vibrator</name>
        <version>1</version>
        <fqname>IVibrator/default</fqname>
    </hal>
```

Otherwise, they should register an AIDL service normally. When running VTS tests, it's expected that all declared AIDL HALs are available.

Note: On devices running Android 11 and lower, don't include <version>.
Write an AIDL client
AIDL clients must declare themselves in the compatibility matrix, for example:

```
    <hal format="aidl" optional="true">
        <name>android.hardware.vibrator</name>
        <version>1-2</version>
        <interface>
            <name>IVibrator</name>
            <instance>default</instance>
        </interface>
    </hal>
```

Note: On devices running Android 11 and lower, don't include <version>.
Convert an existing HAL from HIDL to AIDL
Use the hidl2aidl tool to convert a HIDL interface to AIDL.

hidl2aidl features:

Create AIDL (.aidl) files based on the HAL (.hal) files for the given package.
Create build rules for the newly created AIDL package with all backends enabled.
Create translate methods in the Java, CPP, and NDK backends for translating from the HIDL types to the AIDL types.
Create build rules for translate libraries with required dependencies.
Create static asserts to ensure that HIDL and AIDL enumerators have the same values in the CPP and NDK backends.
Follow these steps to convert a package of HAL files to AIDL files:

Build the tool located in system/tools/hidl/hidl2aidl.

Building this tool from the latest source provides the most complete experience. You can use the latest version to convert interfaces on older branches from previous releases:

```
m hidl2aidl
```

Execute the tool with an output directory followed by the package to be converted.

Optionally, use the -l argument to add the contents of a new license file to the top of all generated files. Be sure to use the correct license and date:

```
hidl2aidl -o <output directory> -l <file with license> <package>
```

For example:

```
hidl2aidl -o . -l my_license.txt android.hardware.nfc@1.2
```

Read through the generated files and fix any issues with the conversion:

conversion.log contains any unhandled issues to fix first.
The generated AIDL files might have warnings and suggestions that need action. These comments begin with //.
Clean up and make improvements to the package.
Check the @JavaDerive annotation for features that might be needed, such as toString or equals.
Build only the targets you need:

Disable backends that won't be used. Prefer the NDK backend over the CPP backend; see Build against the AIDL runtime.
Remove translate libraries or any of their generated code that won't be used.
See Major AIDL and HIDL differences:

Using AIDL's built-in Status and exceptions typically improve the interface and remove the need for another interface-specific status type.
AIDL interface arguments in methods aren't @nullable by default like they were in HIDL.
SEPolicy for AIDL HALs
An AIDL service type that's visible to vendor code must have the hal_service_type attribute. Otherwise, the sepolicy configuration is the same as any other AIDL service (though there are special attributes for HALs). Here's an example definition of a HAL service context:

```
    type hal_foo_service, service_manager_type, hal_service_type;
```

For most services defined by the platform, a service context with the correct type is added already (for example, android.hardware.foo.IFoo/default is already marked as hal_foo_service). However, if a framework client supports multiple instance names, additional instance names must be added in device-specific service_contexts files:

```
    android.hardware.foo.IFoo/custom_instance u:object_r:hal_foo_service:s0
```

When you create a new type of HAL, you must add HAL attributes. A specific HAL attribute might be associated with multiple service types (each of which can have multiple instances as just discussed). For a HAL, foo, there is hal_attribute(foo). This macro defines attributes hal_foo_client and hal_foo_server. For a given domain, the hal_client_domain and hal_server_domain macros associate a domain with a given HAL attribute. For example, system server being a client of this HAL corresponds to the policy hal_client_domain(system_server, hal_foo). A HAL server similarly includes hal_server_domain(my_hal_domain, hal_foo).

Typically, for a given HAL attribute, also create a domain like hal_foo_default for reference or example HALs. However, some devices use these domains for their own servers. Distinguishing between domains for multiple servers matters only if there are multiple servers that serve the same interface and need a different permission set in their implementations. In all of these macros, hal_foo isn't an sepolicy object. Instead, this token is used by these macros to refer to the group of attributes associated with a client server pair.

However, so far, hal_foo_service and hal_foo (the attribute pair from hal_attribute(foo)) aren't associated. A HAL attribute is associated with AIDL HAL services using the hal_attribute_service macro (HIDL HALs use the hal_attribute_hwservice macro), for example, hal_attribute_service(hal_foo, hal_foo_service). This means that hal_foo_client processes can get ahold of the HAL, and hal_foo_server processes can register the HAL. The enforcement of these registration rules is done by the context manager (servicemanager).

Service names might not always correspond to HAL attributes, for example, hal_attribute_service(hal_foo, hal_foo2_service). In general, because this implies the services are always used together, you can remove the hal_foo2_service and use hal_foo_service for all of service contexts. When HALs set multiple hal_attribute_service instances, it's because the original HAL attribute name isn't general enough and can't be changed.

Putting this all together, an example HAL looks like this:

```
public/attributes:
// define hal_foo, hal_foo_client, hal_foo_server
hal_attribute(foo)

public/service.te
// define hal_foo_service
type hal_foo_service, hal_service_type, protected_service, service_manager_type

public/hal_foo.te:
// allow binder connection from client to server
binder_call(hal_foo_client, hal_foo_server)
// allow client to find the service, allow server to register the service
hal_attribute_service(hal_foo, hal_foo_service)
// allow binder communication from server to service_manager
binder_use(hal_foo_server)

private/service_contexts:
// bind an AIDL service name to the selinux type
android.hardware.foo.IFooXxxx/default u:object_r:hal_foo_service:s0

private/<some_domain>.te:
// let this domain use the hal service
binder_use(some_domain)
hal_client_domain(some_domain, hal_foo)

vendor/<some_hal_server_domain>.te
// let this domain serve the hal service
hal_server_domain(some_hal_server_domain, hal_foo)
```

Attached extension interfaces
An extension can be attached to any binder interface, whether it's a top-level interface registered directly with service manager or it's a subinterface. When getting an extension, you must confirm the type of the extension is as expected. You can set extensions only from the process serving a binder.

Use attached extensions whenever an extension modifies the functionality of an existing HAL. When entirely new capability is needed, this mechanism isn't necessary, and you can register an extension interface with the service manager directly. Attached extension interfaces make the most sense when they're attached to subinterfaces, because these hierarchies can be deep or multi-instanced. Using a global extension to mirror the binder interface hierarchy of another service requires extensive bookkeeping to provide equivalent capabilities to directly attached extensions.

To set an extension on a binder, use the following APIs:

NDK backend: AIBinder_setExtension
Java backend: android.os.Binder.setExtension
CPP backend: android::Binder::setExtension
Rust backend: binder::Binder::set_extension
To get an extension on a binder, use the following APIs:

NDK backend: AIBinder_getExtension
Java backend: android.os.IBinder.getExtension
CPP backend: android::IBinder::getExtension
Rust backend: binder::Binder::get_extension
You can find more information for these APIs in the documentation of the getExtension function in the corresponding backend. An example of

how to use extensions is in hardware/interfaces/tests/extension/vibrator.

Major AIDL and HIDL differences
When using AIDL HALs or using AIDL HAL interfaces, be aware of the differences compared to writing HIDL HALs.

The AIDL language's syntax is closer to Java. HIDL syntax is similar to C++.
All AIDL interfaces have built-in error statuses. Instead of creating custom status types, create constant status ints in interface files and use EX_SERVICE_SPECIFIC in the CPP and NDK backends and ServiceSpecificException in the Java backend. See Error handling.
AIDL doesn't automatically start thread pools when binder objects are sent. You must start them manually (see Thread management).
AIDL doesn't abort on unchecked transport errors (HIDL Return aborts on unchecked errors).
AIDL can declare only one type per file.
AIDL arguments can be specified as in, out, or inout in addition to the output parameter (there are no synchronous callbacks).
AIDL uses fd as the primitive type instead of handle.
HIDL uses major versions for incompatible changes and minor versions for compatible changes. In AIDL, backward-compatible changes are done in place. AIDL has no explicit concept of major versions; instead, this is incorporated into package names. For example, AIDL might use the package name bluetooth2.
AIDL doesn't inherit realtime priority by default. The setInheritRt function must be used per binder to enable realtime priority inheritance.
Vendor Test Suite (VTS) Tests for HALs
Android relies on the Vendor Test Suite (VTS) to verify expected HAL implementations. VTS helps ensure Android can be backward compatible with old vendor implementations. Implementations failing VTS have known compatibility issues that could prevent them from working with future versions of the OS.

There are two major parts of VTS for HALs.

1. Verify HALs on the device are known and expected by Android
This set of tests can be found in test/vts-testcase/hal/treble/vintf. These tests are responsible for verifying:

Every @VintfStability interface that is declared in a VINTF manifest is frozen at a known released version. This ensures both sides of the interface agree on the exact definition of that version of the interface. This is necessary for basic operation.
All HALs that are declared in a VINTF manifest are available on that device. Any client with sufficient permissions to use a declared HAL service must be able to get and use those services at any time.
All HALs that are declared in a VINTF manifest are serving the version of the interface that they declare in the manifest.
There are no deprecated HALs being served on a device. Android drops support for lower versions of HAL interfaces as described in FCM lifecycle.
Required HALs are present on the device. Some HALs are required for Android to work properly.
2. Verify the expected behavior of each HAL
Each HAL interface has its own VTS tests to verify the expected behavior from its clients. The test cases run against every instance of a declared HAL interface and enforce specific behavior based on the version of the interface that's implemented.

These tests attempt to cover every aspect of the HAL implementation that the Android framework relies on, or might rely on in the future.

These tests include verifying support of features, error handling, and any other behavior a client might expect from the service.

VTS milestones for HAL development
VTS tests are expected to be kept up to date when creating or modifying Android's HAL interfaces.

VTS tests must be finished and ready to verify vendor implementations before they're frozen for Android Vendor API releases. They must be ready before the interfaces are frozen so developers can create their implementations, verify them, and provide feedback to the HAL interface developers.

VTS on Cuttlefish
When hardware isn't available, Android uses Cuttlefish as a development vehicle for HAL interfaces. This allows for scalable VTS and integration testing of Android.

hal_implementation_test tests that Cuttlefish has implementations of the latest HAL interface versions to make sure Android is ready to handle the new interfaces and the VTS tests are ready to test the new vendor implementations as soon as new hardware and devices are available.

Was this helpful?

Here's a **simple explanation** of the **AIDL for HALs** documentation:

🔄 Why use AIDL for HALs instead of HIDL?

✅ AIDL is now preferred because:

- It existed **before HIDL** and is used across Android (framework, apps, etc.).
- AIDL now **supports stable versioning**, so it's safe for long-term system/vendor communication.
- It has a **simpler versioning model**, easier updates, and better integration.

📦 When to use AIDL HALs

Use AIDL HAL if:

- You're writing a **new HAL**.
- Or you're converting a **HIDL HAL to AIDL** (if upstream HAL doesn't force you to use HIDL).

🔐 Stable AIDL for system ↔ vendor

If a HAL crosses partition boundaries (e.g., from **system** to **vendor**), it **must**:

- Be marked as **`@VintfStability`** in AIDL.
- Be declared in the **VINTF manifest** (**`manifest.xml`**).
- Use **`"stability": "vintf"`** in **`aidl_interface`** rule in **`Android.bp`**.

🧰 Key Benefits of AIDL over HIDL

Feature	AIDL	HIDL
Syntax	Java-like	C++-like
Built-in error handling	✅ (status, exceptions)	❌
Easy versioning	✅ (in-place edits)	❌ (needs new versions)
Thread management	Manual	Automatic
Realtime priority	Optional	Default
Extensions	Runtime-attached	Compile-time only

🧱 AIDL Interface Versioning

- **@VintfStability**: Marks interface as stable.
- **Freezing**: Once stable, use **`m my.hal-freeze-api`** to lock a version (like V1).
- Multiple versions are stored under **`aidl_api/my.hal/V1`**, **`V2`**, etc.

🧩 AIDL Backends

Backend	Use Case	Notes
Java	Apps, system services	Can't use on vendor side (esp. post-API 34)
NDK	HALs in vendor partition	Use **`libbinder_ndk`**
CPP	Deprecated for new use	Disable in **`Android.bp`** if not needed

🧩 AIDL HAL Folder Structure

Location	Purpose
`hardware/interfaces/aidl`	HALs by hardware vendors
`frameworks/hardware/interfaces/aidl`	Higher-level interfaces
`system/hardware/interfaces/aidl`	Low-level system HALs

✏️ Creating AIDL HALs

Example snippet of **`Android.bp`**:

```bp
aidl_interface {
    name: "android.hardware.vibrator",
    srcs: ["IVibrator.aidl"],
    stability: "vintf",
    backend: {
        cpp: { enabled: false }, // disable CPP backend
    },
}
```

And VINTF **`manifest.xml`**:

```xml
<hal format="aidl">
    <name>android.hardware.vibrator</name>
    <version>1</version>
    <fqname>IVibrator/default</fqname>
</hal>
```

🔌 Extensions with `ParcelableHolder`

To **extend** AOSP parcelables without breaking:

- Use **`ParcelableHolder`** field in the original parcelable.
- Add custom parcelables in vendor code.
- Access them using **`.setParcelable()`** / **`.getParcelable()`** in Java/C++/NDK/Rust.

🔒 SELinux (SEPolicy) for AIDL HALs

Define:

```te
type hal_foo_service, hal_service_type, service_manager_type;
hal_attribute(foo)  // defines hal_foo_client, hal_foo_server
hal_attribute_service(hal_foo, hal_foo_service)
```

Grant access:

```te
hal_client_domain(system_server, hal_foo)
hal_server_domain(vendor_foo_server, hal_foo)
```

🔁 Convert HIDL to AIDL using `hidl2aidl`

1. Build:

```bash
m hidl2aidl
```

2. Run:

```bash
hidl2aidl -o . -l my_license.txt android.hardware.nfc@1.2
```

3. Fix issues in generated AIDL files (**`conversion.log`**).
4. Freeze new interface with **`m my.hal-freeze-api`**.

✅ VTS Requirements

Two major goals:

1. Ensure correct versioning and availability:
   - HALs declared in VINTF must be **frozen**, **present**, and **serve correct versions**.
2. Validate HAL behavior:
   - Tests for correct behavior, supported features, and error handling.

🚀 Summary

Topic	AIDL
Replaces HIDL	✅ Yes, in most cases
Needs **`@VintfStability`**	✅ For stable interfaces
Must be declared in VINTF	✅ If crossing system/vendor
Better versioning	✅
Supports extensions	✅ via **`ParcelableHolder`** and **`setExtension()`**
Prefer backend	✅ NDK
VTS required	✅ Yes

Let me know if you'd like diagrams or a full working example of an AIDL HAL in AOSP.

9. ↪
🧱 AIDL Interface Versioning --------------------------- * **@VintfStability**: Marks interface as stable. * **Freezing**: Once stable, use `m my.hal-freeze-api` to lock a version (like V1). * Multiple versions are stored under `aidl_api/my.hal/V1`, `V2`, etc.
explain

Sure! Let me explain **AIDL interface versioning** in **simple words**:

✅ What is `@VintfStability`?

This is an **annotation** in your `.aidl` file that tells Android:

> "This interface is **stable** and can be safely used between different partitions like system ↔ vendor."

You add this line above your interface or parcelable:

```aidl
@VintfStability
interface IFoo { ... }
```

Once this is added, Android treats this interface as **versioned and safe** to ship.

❄️ What is "freezing" an AIDL interface?

Once your interface is **ready for public or long-term use**, you **freeze** it to lock its version forever.

You do this by running a build command:

```bash
m my.hal-freeze-api
```

This creates a **snapshot** of your current AIDL definition so that:

- Android can **track which version you're using** (V1, V2, etc.).
- Future changes to the interface are **explicit and controlled**.

📁 Where are versions stored?

After freezing, Android creates folders like:

```pgsql
aidl_api/my.hal/V1/        ← Your first stable version
aidl_api/my.hal/V2/        ← After you make backward-compatible changes and re-freeze
```

Each version folder contains `.aidl` snapshots that define what the interface looked like at that version.

🔄 When do I need to create a new version?

You must freeze a **new version (V2, V3, etc.)** when:

- You **add new methods**.
- You **change existing structures** in a **backward-compatible** way.

If the change is **not compatible**, you need to:

- Create a new interface package, like `android.hardware.foo2`.

🔐 Why is this important?

This versioning system ensures that:

- System and vendor partitions **agree** on what a HAL interface means.
- The interface remains **compatible** across Android versions.
- The **VTS test** can verify that your device uses a known, frozen interface.

✅ Summary

Term	Meaning
`@VintfStability`	Marks interface as stable and versioned
`m my.hal-freeze-api`	Command to lock the current version
`aidl_api/.../V1`	Where frozen interface versions live
V2, V3	Created when changes are made and re-frozen

Let me know if you want a step-by-step guide to freeze and test an AIDL HAL in AOSP!

10. but how versioning is working here explain with example

Great question! Let's walk through **how AIDL versioning works in practice**, step-by-step, with a clear **example**.

🔧 Scenario: You're creating an AIDL HAL for a device

Let's say you're defining a HAL interface for a **LED controller**:

✅ Step 1: Initial interface (version 1)

File: `hardware/interfaces/led/aidl/android/hardware/led/ILed.aidl`

```aidl
@VintfStability
interface ILed {
    void turnOn();
    void turnOff();
}
```

Now, you **freeze version 1** using:

```bash
m android.hardware.led-freeze-api
```

Android will generate:

```bash
aidl_api/android.hardware.led/V1/ILed.aidl
```

This version (V1) is now considered **stable and frozen**. System and vendor will agree on this definition.

✅ Step 2: Add a new method → make version 2

You decide to **add a new method** to blink the LED:

```aidl
@VintfStability
interface ILed {
    void turnOn();
    void turnOff();
    void blink(int durationMs);  // newly added method
}
```

This change is **backward compatible**, so you just **re-freeze**:

```bash
m android.hardware.led-freeze-api
```

Now, this will create:

```bash
aidl_api/android.hardware.led/V2/ILed.aidl
```

So now, Android knows:

- V1 has `turnOn()` and `turnOff()`
- V2 adds `blink(int durationMs)`

🧠 How Android uses this version info?

1. 📱 **Device declares in VINTF manifest**:

```xml
<hal format="aidl">
    <name>android.hardware.led</name>
    <version>2</version> <!-- Uses version 2 -->
    <fqname>ILed/default</fqname>
</hal>
```

2. ✅ During **VTS testing**, Android checks:
   - Whether version **2** exists in **`aidl_api/.../V2`**
   - Whether **`ILed`** service is **really serving V2**, and not a mismatch
3. 🔄 If a client (like system server) requests **`ILed/default`**, Android ensures it's talking to a version-2 compatible implementation.

🔄 What happens in case of breaking changes?

AIDL doesn't support breaking changes (e.g., removing a method or changing method arguments) **within the same interface**.

So if you need incompatible changes, you do this:

```aidl
// New interface in a new package (e.g. "led2")
package android.hardware.led2;

@VintfStability
interface ILedV2 {
    void turnOn();
    void turnOff();
    void setBrightness(int level);
}
```

Then declare that as a **new HAL** in the manifest:

```xml
<hal format="aidl">
    <name>android.hardware.led2</name>
    <version>1</version>
    <fqname>ILedV2/default</fqname>
</hal>
```

💡 Summary table

Situation	What you do	What happens
Add method or field	Update **`.aidl`**, then freeze again	New version (V2, V3...) created
Breaking change	Create new package like **`led2`**	Keeps old clients safe
Build-time freeze	**`m <your-hal-name>-freeze-api`**	Stores **`.aidl`** in **`aidl_api/...`**
Runtime	Declared in VINTF manifest	Framework can verify compatibility

---

Would you like a **real frozen `.aidl_api/.../V1` file example** or **walkthrough of using it in C++/NDK code**?

11. ↪
🧩 AIDL HAL Folder Structure ---------------------------- | Location | Purpose | | --- | --- | | `hardware/interfaces/aidl` | HALs by hardware vendors | | `frameworks/hardware/interfaces/aidl` | Higher-level interfaces | | `system/hardware/interfaces/aidl` | Low-level system HALs |
explain in details with example

Let's walk through **AIDL versioning for HALs** in **detailed steps with examples**, starting from interface creation to how versioning works, how freezing works, and how Android ensures compatibility across system and vendor boundaries.

📌 Part 1: Create Your AIDL HAL Interface

🧪 Example Use Case: LED Hardware Control

Let's create an AIDL HAL interface for controlling a device's LED.

🔹 Step 1: Define AIDL HAL Interface

🗂️ Folder:
`hardware/interfaces/led/aidl/android/hardware/led/ILed.aidl`

📄 File: `ILed.aidl`

```aidl
package android.hardware.led;

@VintfStability
interface ILed {
    void turnOn();
    void turnOff();
}
```

🧩 Folder Structure

Folder	Purpose
`hardware/interfaces/led/aidl`	Your AIDL HAL source
`hardware/interfaces/led/aidl/android/hardware/led/ILed.aidl`	AIDL interface
`aidl_api/android.hardware.led/V1/`	Frozen API of version 1 after running freeze script

📌 Part 2: Enable Versioning by Freezing the API

Run the following command:

```bash
m android.hardware.led-freeze-api
```

This generates:

🗂️ `hardware/interfaces/led/aidl_api/android.hardware.led/V1/`

With contents like:

```bash
ILed.aidl
.hash
```

✅ **Frozen AIDL** = The system now knows this is version 1.

📌 Part 3: Add a Feature (Backward-Compatible Change)

Let's say you want to add a new method to blink the LED.

🔹 Step 1: Modify `ILed.aidl`

```aidl
package android.hardware.led;

@VintfStability
interface ILed {
    void turnOn();
    void turnOff();
    void blink(int milliseconds); // 🔄 New method
}
```

🔹 Step 2: Freeze Again to Generate Version 2

```bash
m android.hardware.led-freeze-api
```

Creates:
🗂️ **`hardware/interfaces/led/aidl_api/android.hardware.led/V2/ILed.aidl`**

Now, the system supports **both V1 and V2** of **`ILed`**.

📌 Part 4: Use in VINTF Manifest

Vendor side (declares HAL provided):

📄 **`vendor/manifest.xml`**

```xml
<hal format="aidl">
    <name>android.hardware.led</name>
    <version>2</version>
    <fqname>ILed/default</fqname>
</hal>
```

System side (declares client):

📄 **`system/compatibility_matrix.xml`**

```xml
<hal format="aidl" optional="false">
    <name>android.hardware.led</name>
    <version>1-2</version>
    <interface>
        <name>ILed</name>
        <instance>default</instance>
    </interface>
</hal>
```

📌 Now the system expects any **`ILed/default`** HAL to implement **at least version 1** and up to **version 2**.

📌 Part 5: How Android Handles This?

✅ During VTS Test:

- Checks that version 2 exists at **`aidl_api/android.hardware.led/V2`**
- Ensures that service declared in VINTF (**`ILed/default`**) **really implements V2**
- Verifies all declared methods in V2 are correctly handled by vendor service
- Confirms no deprecated versions (like V0) are used

📌 Part 6: What if Breaking Change Needed?

Say, in next year, you want to change `blink(int)` to `blink(int, int)` (a breaking change).

You cannot modify the existing interface directly. Instead:

🔄 Create new interface under a new package name:

📄 **`hardware/interfaces/led2/aidl/android/hardware/led2/ILed.aidl`**

```aidl
package android.hardware.led2;

@VintfStability
interface ILed {
    void turnOn();
    void turnOff();
    void blink(int durationMs, int intensity);
}
```

🔐 Now freeze:

```bash
m android.hardware.led2-freeze-api
```

📄 New VINTF:

```xml
<hal format="aidl">
    <name>android.hardware.led2</name>
    <version>1</version>
    <fqname>ILed/default</fqname>
</hal>
```

🧠 **Different HAL name** = No compatibility issue with older **`android.hardware.led`**.

📌 Summary Table

Action	What Happens
Add a method to existing interface	Create new version (V2, V3...) by **`-freeze-api`**
Breaking change (e.g., method signature change)	Must use new package name (**`led2`**)
Freeze command	Generates **`.aidl`** snapshot under **`aidl_api/...`**
VINTF Manifest	Declares provided or required version
VTS	Verifies actual vs declared version, interface behavior

✅ Final Example Layout

```bash
hardware/
└── interfaces/
    └── led/
        ├── aidl/
        │   └── android/hardware/led/ILed.aidl
        └── aidl_api/
            └── android.hardware.led/
                ├── V1/ILed.aidl
                └── V2/ILed.aidl
```

Would you like a **full AIDL HAL server+client sample in C++ or NDK**, or the **SEPolicy parts for this HAL** next?
Let's break down the **three AIDL interface locations** used in AOSP in detail — including **purpose**, **usage**, and **real examples** — so you understand why different AIDL HALs are placed in different directories:

📌 1. `hardware/interfaces/aidl`

✅ Purpose:

HAL interfaces **implemented by hardware vendors**. These are typically **vendor-specific or standard hardware abstraction** interfaces like for Bluetooth, Vibrator, Camera, Display, Sensors, Audio, etc.

🏗️ Who owns it?

- Mostly **Google** and **hardware vendors (OEMs/ODMs)**.
- HAL implementations reside in **`vendor/`** or **`hardware/`** partitions.

📂 Directory Structure Example:

```swift
hardware/interfaces/vibrator/aidl/android/hardware/vibrator/IVibrator.aidl
```

📄 Example AIDL HAL Interface:

```aidl
// hardware/interfaces/vibrator/aidl/android/hardware/vibrator/IVibrator.aidl

package android.hardware.vibrator;

@VintfStability
interface IVibrator {
    void on(int durationMs);
    void off();
}
```

📑 Use Case:

Used by the **Android framework (system)** to control hardware-specific functionality (like vibration).

📦 VINTF Manifest Entry:

```xml
<hal format="aidl">
    <name>android.hardware.vibrator</name>
    <version>1</version>
    <fqname>IVibrator/default</fqname>
</hal>
```

📌 2. `frameworks/hardware/interfaces/aidl`

✅ Purpose:

Contains **high-level framework-to-hardware interfaces**, not directly controlling hardware but **providing logical behavior** for HALs or coordinating them.

Think of it as HALs that are **"hardware-aware" but closer to the Android Framework**.

🏗️ Who owns it?

- **AOSP framework developers**.
- Vendors may implement these HALs but they don't usually interact with the hardware directly.

📂 Directory Structure Example:

```swift
frameworks/hardware/interfaces/sensor/aidl/android/frameworks/sensor/ISensorManager.aidl
```

📄 Example AIDL Interface:

```aidl
// frameworks/hardware/interfaces/sensor/aidl/android/frameworks/sensor/ISensorManager.aidl

package android.frameworks.sensor;

@VintfStability
interface ISensorManager {
    List<String> getAvailableSensors();
    boolean enableSensor(String name);
}
```

📑 Use Case:

- Enables abstraction over **multiple hardware HALs**.
- Could interact with **`android.hardware.sensors`** HAL internally.
- Used in frameworks like SensorService.

📌 3. `system/hardware/interfaces/aidl`

✅ Purpose:

Contains **low-level system HALs** (such as kernel driver communication) or **core platform services** that are tightly bound to the **OS internals**, not tied to specific hardware devices.

These are often closer to the **system server** or **Android's boot chain**, and sometimes help initialize or configure the system.

🏗️ Who owns it?

- **Android OS developers** (AOSP team).

📂 Directory Structure Example:

```swift
system/hardware/interfaces/powerstats/aidl/android/system/powerstats/IPowerStats.aidl
```

📄 Example AIDL Interface:

```aidl
// system/hardware/interfaces/powerstats/aidl/android/system/powerstats/IPowerStats.aidl

package android.system.powerstats;

@VintfStability
interface IPowerStats {
    float getAveragePowerUsage();
}
```

📑 Use Case:

- Fetch internal platform/system-level metrics.
- Works with data from **`sysfs`**, **`procfs`**, etc., not just hardware HALs.
- **Used by system_server**, not by applications.

✅ Comparison Summary Table

Location	Owner	Purpose	Example AIDL	Accessed By
`hardware/interfaces/aidl`	Vendors / Google	Low-level HALs for **direct hardware** (e.g., Camera, Vibrator)	**`IVibrator`**	Android framework
`frameworks/hardware/interfaces/aidl`	AOSP Framework	**High-level HALs**, logic abstraction above hardware	**`ISensorManager`**	Android system services
`system/hardware/interfaces/aidl`	AOSP System	Low-level **platform HALs** or system metrics (e.g., power stats, debug data)	**`IPowerStats`**	system_server / init

🧩 Where Should I Place My AIDL HAL?

Use Case	Folder
You're creating a HAL for hardware like LED, Audio, Display	`hardware/interfaces/aidl`
You want to abstract multiple hardware HALs into a manager	`frameworks/hardware/interfaces/aidl`
You're creating a HAL that interacts with system internals like logs, power stats	`system/hardware/interfaces/aidl`

Would you like a **working AIDL HAL example from `hardware/interfaces/`** with full Android.bp and service implementation next?

> 12. ↪
> `frameworks/hardware/interfaces/aidl`
> but these have to interact with driver at the end na?

Great question — and yes, you're absolutely right to wonder:

> **"Don't they all eventually talk to drivers anyway?"**

Let me explain how it works **specifically for AIDL HALs in `frameworks/hardware/interfaces/aidl`**, and **why they are *not always* directly tied to hardware drivers.**

🧠 Key Understanding

> ✅ **Not all HALs need to directly interact with kernel drivers.**

Some **mediate**, **aggregate**, or **extend functionality** of other lower-level HALs (like those in `hardware/interfaces`) — essentially acting as "middlemen" between framework and true hardware HALs.

🧩 So what do HALs in `frameworks/hardware/interfaces/aidl` do?

They typically:

1. ✅ **Aggregate or manage multiple hardware-level HALs**
2. ✅ **Expose logical APIs built on top of lower HALs**
3. ✅ **Provide backward compatibility layers**
4. ✅ **Talk to `hardware/interfaces` HALs, not directly to the Linux kernel**

🔁 Example Architecture (Layered)

Here's how the layers look:

```css
[ App / Java Framework ]
          ↓
[ System Service ]
          ↓
[ HAL @ frameworks/hardware/interfaces/aidl ]
          ↓
[ HAL @ hardware/interfaces/aidl ]
          ↓
[ Kernel Driver (via ioctl/sysfs/netlink etc.) ]
```

🔍 Case Study: SensorManager

Layer	Details
`frameworks/hardware/interfaces/aidl/android/frameworks/sensor/ISensorManager.aidl`	AIDL HAL providing unified access to multiple sensors.
Talks to →	`hardware/interfaces/sensors/aidl/android/hardware`
Which talks to →	Sensor HAL module (e.g., `/vendor/lib/hw/sensors.qcom.s`
Which talks to →	Linux kernel drivers (`/dev/iio`, `/sys/class/sensors/...`)

So yes — eventually the path leads to a driver, **but the AIDL HAL in `frameworks/hardware/` doesn't directly touch the driver** — it **talks to the HAL in `hardware/interfaces`** which does.

🧭 Why separate it this way?

✅ Modularity

- Google wants to **separate platform logic** (in `frameworks/`) from **vendor-specific hardware code** (`hardware/` and `vendor/`).

✅ Reusability

- A framework-level AIDL HAL can remain stable across devices, while the hardware HAL may vary per SoC/OEM.

✅ Testing and Abstraction

- You can test higher-level HAL logic (e.g., filtering sensor data, managing multiple sensors) without needing hardware.

🧪 When might one directly touch drivers?

If you're writing a HAL in:

- `hardware/interfaces/...` ➜ likely yes (you **implement ioctl/sysfs** calls).
- `frameworks/hardware/interfaces/...` ➜ **probably no** (you use other HALs underneath).
- `system/hardware/interfaces/...` ➜ sometimes yes (e.g., for powerstats, bootcontrol, debug).

🧠 Summary

HAL Location	Direct Driver Access?	Purpose
`hardware/interfaces/aidl`	✅ Yes	Talks to drivers (via ioctl, sysfs, etc.)
`frameworks/hardware/interfaces/aidl`	❌ No (uses lower HALs)	Aggregates/extends lower HALs
`system/hardware/interfaces/aidl`	⚠️ Maybe	System metrics, boot services

Let me know if you want a **real layered HAL flow example**, like a full `SensorService → ISensorManager → ISensors → driver` call chain, or a sample AIDL HAL from each layer.

13. ↪
system/hardware/interfaces/
same these as well an?

Yes — let's now explain `system/hardware/interfaces/` HALs and whether **they also interact with drivers directly** or not.

✅ Short Answer

- **Some** HALs in `system/hardware/interfaces/` **do interact with drivers**,
- but **most do not** — they usually:

- Expose **low-level services** needed by the system (like power stats, reboot reason, suspend info),
- Or talk to **other HALs** that deal with hardware.

So, **they are a mix** — **some touch drivers**, others **mediate between subsystems**.

🧩 What's special about `system/hardware/interfaces/`?

This location is for **system-level low-level services** that are **tightly coupled with the platform**, but **not part of hardware abstraction layers for OEMs**.

Examples include:

Interface	Purpose	Direct driver access?
`SuspendControlService`	Tells system when to suspend/resume	✅ Sometimes
`PowerStats`	Tracks power usage per subsystem	✅ Yes (reads from `/sys/...`)
`RebootEscrow`	Stores secrets across reboots	❌ No (talks to keystore etc.)
`Thermal`	Reports thermal info	✅ Yes (reads `/sys/class/thermal/...`)
`Health`	Reports battery/charging info	✅ Yes (interacts with charger driver)

🔄 A Typical Call Flow (Example)

Say we have `system/hardware/interfaces/powerstats/aidl`.

The flow could look like:

```css
[ System Service / Framework ]
          ↓
[ AIDL HAL @ system/hardware/interfaces/powerstats/ ]
          ↓
[ Reads from /sys/class/power_supply or debugfs ]
          ↓
[ Kernel driver ]
```

So yes — **this is an example where the AIDL HAL directly reads from kernel driver nodes** (e.g., `/sys/class/...`, `/proc`, `/dev`).

🧠 Summary Table

AIDL HAL Location	Purpose	Usually Talks to Drivers?
`frameworks/hardware/interfaces/`	High-level platform services	❌ No (talks to other HALs)
`hardware/interfaces/`	Vendor-specific hardware HALs	✅ Yes (interacts with kernel)
`system/hardware/interfaces/`	Low-level system metrics / info	⚠️ Sometimes Yes

💡 Rules of Thumb

- If the HAL **relates to a hardware component or sensor**, and it's in **`hardware/interfaces`**, it **talks to drivers**.
- If it's in **`frameworks/hardware/interfaces`**, it's probably just **a logical layer** or **mediator**.
- If it's in **`system/hardware/interfaces`**, check the specific HAL — **some read system files directly**, others **don't touch drivers at all**.

Would you like a full example like **`powerstats`** or **`thermal`** HAL from **`system/hardware/interfaces`** showing how it reads from **`/sys`**?
Let's break down how **`ParcelableHolder`** works using a simple, clear example — especially for when a **vendor (OEM)** wants to **extend AOSP-defined parcelables** without causing compatibility or merge issues.

🧠 Why use ParcelableHolder?

Imagine AOSP defines a parcelable like:

```aidl
// In AOSP (framework)
parcelable BatteryStatus {
    int level;
    boolean charging;
}
```

Now suppose **Samsung** wants to add extra data like **`temperature`**, **`health`**, etc. If they directly modify the AOSP file:

```aidl
// ❌ DON'T DO THIS - This breaks compatibility
parcelable BatteryStatus {
    int level;
    boolean charging;
    float temperature;      // added by OEM
    String batteryHealth;   // added by OEM
}
```

💥 This is **not allowed**. It causes **conflicts** during Android version upgrades and breaks **Generic System Image (GSI)** compatibility.

✅ The correct solution: Use `ParcelableHolder`

AOSP provides a hook like:

```aidl
// AOSP defines this
parcelable BatteryStatus {
    int level;
    boolean charging;
    ParcelableHolder extension;  // Placeholder for vendor extensions
}
```

Then OEM (Samsung, etc.) defines their own parcelable:

```aidl
// In vendor-specific code
parcelable OemBatteryExtension {
    float temperature;
    String batteryHealth;
}
```

🔧 How to Use It

☕ Java Example

```java
// Set the OEM extension
BatteryStatus bs = new BatteryStatus();
bs.level = 90;
bs.charging = true;

OemBatteryExtension ext = new OemBatteryExtension();
```

```
ext.temperature = 38.5f;
ext.batteryHealth = "Good";

bs.extension.setParcelable(ext);  // 📦 Store vendor extension

// Later: retrieve it
OemBatteryExtension ext2 = bs.extension.getParcelable(OemBatteryExtension.class);
Log.d("Temp", "Battery temperature = " + ext2.temperature);
```

⚙️ C++ Example (NDK)

```cpp
BatteryStatus bs;
bs.level = 90;
bs.charging = true;

OemBatteryExtension ext;
ext.temperature = 38.5f;
ext.batteryHealth = "Good";

bs.extension.setParcelable(ext);

// Later
std::optional<OemBatteryExtension> read_ext;
bs.extension.getParcelable(&read_ext);

if (read_ext) {
    LOG(INFO) << "Battery health: " << read_ext->batteryHealth;
}
```

🔐 Key Benefit

- **No need to fork AOSP parcelables**.
- **System remains compatible with GSI**.
- **Your OEM-specific features still work!**

📦 Real-world Analogy

Think of **`ParcelableHolder`** like a **"pocket"** in a standard uniform:

> 📦 AOSP gives you a jacket with a sewn-in pocket (**`ParcelableHolder`**).
> 🎨 As a vendor, you're allowed to put **your own items** in that pocket, but **you can't change the jacket itself**.

📘 Summary

Feature	Description
`ParcelableHolder`	A flexible container to hold vendor-defined parcelables
Purpose	To **extend AOSP parcelables** without editing their definitions
How to use	AOSP defines **`extension: ParcelableHolder`**, vendor defines their own parcelable and uses **`set/getParcelable()`**
Languages supported	Java, C++, NDK, Rust
Avoids	Merge conflicts, GSI failures, and tight coupling

Let me know if you want a full Android project with **`BatteryStatus`** and **`OemBatteryExtension`** implemented as an example.
Great! You're looking at how to define **SELinux (SEPolicy)** rules for **AIDL HALs**.

Let's break this down clearly, line by line, and explain how to correctly set up a secure SELinux policy for an AIDL-based HAL.

🔧 Step-by-Step Breakdown

✅ 1. Define the SELinux service type

```te
type hal_foo_service, hal_service_type, service_manager_type;
```

- This declares a **new binder service type** named **`hal_foo_service`**.
- **`hal_service_type`** tells SELinux this is a HAL service.
- **`service_manager_type`** means this service will be **registered in the Binder service manager** (**`servicemanager`**).

👉 Think of this like:

> "I'm creating a service called **`hal_foo_service`** that HAL clients can look up via Binder."

✅ 2. Define the HAL attribute and access groups

```te
hal_attribute(foo)
```

This macro defines:

- **`hal_foo_client`** → domains that are **allowed to use (call)** the HAL
- **`hal_foo_server`** → domains that are **allowed to implement (serve)** the HAL

So, this single line expands to define **three SELinux types**:

- **`hal_foo`**
- **`hal_foo_client`**
- **`hal_foo_server`**

These are used to **group permissions** cleanly.

✅ 3. Associate the service name with the HAL attribute

```te
hal_attribute_service(hal_foo, hal_foo_service)
```

This **links** the logical HAL attribute (**`hal_foo`**) to the **actual binder service type** (**`hal_foo_service`**).

📌 Without this line, SELinux won't allow binder clients/servers to access **`hal_foo_service`**, even if you define the domains.

✅ 4. Allow system to act as a client

```te
hal_client_domain(system_server, hal_foo)
```

This **grants permission** for **`system_server`** to use the HAL:

- Allows it to **find and connect** to **`hal_foo_service`**
- Allows it to **send binder calls** to the HAL server

✅ 5. Allow a vendor process to serve the HAL

```te
hal_server_domain(vendor_foo_server, hal_foo)
```

This **grants permission** for **`vendor_foo_server`** to:

- **Register** the HAL under **`hal_foo_service`**

- **Receive binder calls** from clients

🧱 Example: Full Setup

Let's say you're creating a HAL called **`android.hardware.foo.IFoo/default`**.

🔹 Define the SELinux types:

```te
// public/service.te
type hal_foo_service, hal_service_type, service_manager_type;

// public/attributes
hal_attribute(foo)

// public/hal_foo.te
hal_attribute_service(hal_foo, hal_foo_service)
```

🔹 Grant access:

```te
// Allow system_server to be the client
hal_client_domain(system_server, hal_foo)

// Allow your HAL process (e.g., foo_hal) to be the server
hal_server_domain(foo_hal, hal_foo)
```

🔹 Bind AIDL name to SELinux context:

```arduino
// private/service_contexts
android.hardware.foo.IFoo/default  u:object_r:hal_foo_service:s0
```

✅ Result

- **`system_server`** can **find and connect** to the **`android.hardware.foo.IFoo/default`** service.
- **`foo_hal`** (your HAL binary) can **register and serve** the HAL over Binder.
- SELinux **enforces** that **no other process** can register or use this HAL unless explicitly allowed.

🧠 Tip: Naming Convention

- Use **`hal_<halname>_service`** as the service type
- Use **`hal_<halname>_client`** and **`hal_<halname>_server`** for grouping domains
- Keep all of this in matching files for maintainability

Would you like a real example for a specific HAL like **`vibrator`**, **`health`**, or a custom one (e.g., **`led`**)? I can show all **`.te`**, **`service_contexts`**, and VINTF manifest examples.

14. Run AIDL services dynamically

bookmark_border
Starting in Android 11, native AIDL services running in the system partition can be started and stopped dynamically as they are needed. Dynamic services start when they are first requested and automatically stop when they are no longer in use.

Warning: Due to Java garbage collection, Java-based services might take a long time to shut down. If this is problematic, you might want to avoid Java client code for dynamic services.
Services that can run dynamically
This feature is available only for native services whose lifecycles can be controlled by init and servicemanager. Services within app packages are not supported and should use bound services instead.

Dynamic shutdown works by shutting down the process in which the service runs. If multiple services exist in the same process, all of them must be registered as dynamic to be compatible with this feature. That process will then shut down when all of the services are unused.

Configure a service's init .rc file
To run a service dynamically, add the following options to the service's init .rc file after the leading service <name> <cmd> line.

interface aidl serviceName
disabled
oneshot
These options do the following:

interface aidl serviceName: Allows servicemanager to find the service. If the service uses multiple interfaces, declare each interface on its own line. These names must be exactly what servicemanager expects and may differ from the process name.
disabled: Prevents the service from automatically starting at boot.
oneshot: Prevents the service from automatically restarting each time it is stopped.
For more information, see the Android Init Language Readme in AOSP.

Examples:

apexd
gsid
Register a service
Each service is created and registered with servicemanager. Registration often occurs in a file named main.cpp, but the implementation can vary. The registration usually looks something like this:

using android::defaultServiceManager;

defaultServiceManager()->addService(serviceName, service);
Registration is sometimes abstracted by BinderService::publish or BinderService::instantiate, which call the above code.

To register a service as dynamic, replace its registration code with the following:

#include <binder/LazyServiceRegistrar.h>

using android::binder::LazyServiceRegistrar;

auto lazyRegistrar = LazyServiceRegistrar::getInstance();
lazyRegistrar.registerService(service, serviceName);
servicemanager communicates with LazyServiceRegistrar to shut down services based on their reference counts.

Examples:

apexservice
gsi_service
Configure AIDL service clients
Get the service
To retrieve a lazy service, the service must be started and then retrieved. Calling getService on the service manager will start the service, but usually, you want to get the service as soon as it is available, and waitForService variants should be used. See backend-specific documentation on

how to use these.

Release the service
Dynamic shutdown is based on reference counting, so clients must not hold onto the service when it is not in use.

Examples:

ApexManager
libgsid
Temporarily disable shutdown
If you want a service to run independently until certain tasks are complete and then switch to dynamic behavior, you can use LazyServiceRegistrar::forcePersist to toggle dynamic shutdown on and off. If this is called from the server side, it should be called before registerService.

Example: apexservice

Here's a **simple explanation** of how **dynamic AIDL services** work in Android 11+ and how to implement them using native (C++ or NDK) code.

🚀 What Are "Dynamic AIDL Services"?

Normally, binder services in Android **start at boot** and **run forever**.
With **dynamic AIDL services**, native services can:

- 🚫 Not start at boot.
- 🟢 Start automatically **only when a client requests them**.
- 💤 Stop automatically **when no one is using them**.

This saves **RAM and power**, especially on embedded or low-power devices.

🧱 Requirements

- Only **native AIDL services** (not Java-based) can use this feature.
- The service must be launched via `init.rc` using `oneshot` + `disabled`.
- Use `LazyServiceRegistrar` in your C++ code to register the service.
- The service must run **in its own process** or with other dynamic services only.

🛠️ Step-by-Step Implementation

📁 1. Add to your `.rc` file (e.g., `vendor/etc/init/myhal.rc`)

```rc
service vendor.myhal /vendor/bin/hw/myhalservice
    interface aidl vendor.myhal.IFoo default
    disabled
    oneshot
    user system
    group system
```

What these mean:

Option	Purpose
`interface aidl vendor.myhal.IFoo default`	Tells **`servicemanager`** this service provides AIDL interface **`vendor.myhal.IFoo/default`**
`disabled`	Don’t start this service at boot
`oneshot`	Stop the service if it crashes (don't restart it automatically)

🧑‍💻 2. Register your service in C++ (main.cpp)

Replace the usual `BinderService::publish()` with this:

```cpp
#include <binder/LazyServiceRegistrar.h>
#include "FooService.h"

using namespace android;

int main() {
    android::ProcessState::initWithDriver("/dev/vndbinder");

    // Get lazy registrar
    auto registrar = binder::LazyServiceRegistrar::getInstance();

    // Create your HAL service
    sp<IFoo> service = new FooService();

    // Register lazily
    registrar.registerService(service, "default");

    // Start binder thread pool
    android::IPCThreadState::self()->joinThreadPool();
}
```

✅ Now the service **won't start until** a client does **`waitForService()`**!

🧑‍💻 3. Client gets the service

Instead of **`getService()`** (which may return null if service isn't ready), use:

```cpp
sp<IFoo> foo = IFoo::fromBinder(
    ndk::SharedRefBase::make<AIBinder>(
        AServiceManager_waitForService("vendor.myhal.IFoo/default")
    )
);
```

This waits for the service to be started **and ready**.

🧼 4. Releasing the Service

💡 The system tracks **reference count** (how many processes are using it).
Once **no one is holding a reference**, the service automatically shuts down.

❗ In your client:

- Don't hold onto **`sp<IFoo>`** forever.
- Let it go when you're done, so the service can shut down.

🔒 Optional: Temporarily prevent auto-shutdown

If your service needs to stay alive during some setup:

```cpp
auto registrar = binder::LazyServiceRegistrar::getInstance();
registrar.forcePersist(true); // disable auto-shutdown temporarily

registrar.registerService(service, "default");

// Later, allow shutdown
registrar.forcePersist(false);
```

🧪 Examples in AOSP

These use dynamic AIDL services:

Service	Path
`apexservice`	**`system/apex/apexd/`**
`gsid`	**`system/gsid/`**

✅ Summary

Feature	Description
`disabled + oneshot`	Service only starts when requested, and shuts down when unused
`LazyServiceRegistrar`	Enables automatic tracking of usage and shutdown
`waitForService()`	Starts the service and waits until it's ready
Reference-based shutdown	When no one is using the service, it exits
Saves power and RAM	Especially useful for infrequently used system services

---

Let me know if you'd like a full AIDL HAL demo (with **`.aidl`**, **`.rc`**, **`main.cpp`**, and a sample client)!

15. Annotations in AIDL

bookmark_border
AIDL supports annotations that give the AIDL compiler extra info about the annotated element, which also affects the generated stub code.

The syntax is similar to that of Java:

```
@AnnotationName(argument1=value, argument2=value) AidlEntity
```

Here, AnnotationName is the name of the annotation, and AidlEntity is an AIDL entity like interface Foo, void method(), or int arg. An annotation is attached to the entity that follows it.

Some annotations can have arguments set inside the parentheses, as shown above. Annotations that don't have an argument don't need the parenthesis. For example:

```
@AnnotationName AidlEntity
```

These annotations are not the same as the Java annotations, although they look very similar. Users can't define custom AIDL annotations; the annotations are all predefined. Some annotations affect only a certain backend and are no-op in other backends. They have different restrictions where they can be attached to.

Here is the list of predefined AIDL annotations:

Annotations	Added in Android version
nullable	7
utf8InCpp	7
VintfStability	11
UnsupportedAppUsage	10
Hide	11
Backing	11
NdkOnlyStableParcelable	14
JavaOnlyStableParcelable	11
JavaDerive	12
JavaPassthrough	12
FixedSize	12
Descriptor	12

Note: The AIDL compiler parses @hide and @deprecated in comments.
nullable
nullable declares that the value of the annotated entity may not be provided.

This annotation can only be attached to method return types, method parameters, and parcelable fields.

```
interface IFoo {
   // method return types
   @nullable Data method();

   // method parameters
   void method2(in @nullable Data d);
}

parcelable Data {
   // parcelable fields
   @nullable Data d;
}
```

Annotations can't be attached to primitive types. The following is an error.

```
void method(in @nullable int a); // int is a primitive type
```

This annotation is no-op for the Java backend. This is because, in Java, all non-primitive types are passed by reference, which could be null.

In the CPP backend, @nullable T maps to std::unique_ptr<T> in Android 11 or lower, and to std::optional<T> in Android 12 or higher.

In the NDK backend, @nullable T always maps to std::optional<T>.

In the Rust backend, @nullable T always maps to Option<T>.

For a list-like type L such as T[] or List<T>, @nullable L maps to std::optional<std::vector<std::optional<T>>> (or std::unique_ptr<std::vector<std::unique_ptr<T>>> in case of the CPP backend for Android 11 or lower).

There is an exception to this mapping. When T is IBinder or an AIDL interface, @nullable is no-op for all backends except for Rust. In other words, both @nullable IBinder and IBinder equally map to android::sp<IBinder>, which is already nullable because it is a strong pointer (CPP reads still enforce nullability, but the type is still android::sp<IBinder>). In Rust, these types are nullable only if annotated with @nullable. They map to Option<T> if annotated.

Beginning with Android 13, @nullable(heap=true) can be used for parcelable fields to model recursive types. @nullable(heap=true) can't be used with method parameters or return types. When annotated with it, the field is mapped to a heap-allocated reference std::unique_ptr<T> in the CPP/NDK backends. @nullable(heap=true) is no-op in the Java backend.

utf8InCpp
utf8InCpp declares that a String is represented in UTF8 format for the CPP backend. As its name indicates, the annotation is a no-op for other backends. Specifically, String is always UTF16 in the Java backend and UTF8 in the NDK backend.

This annotation can be attached anywhere the String type can be used, including return values, parameters, constant declarations, and parcelable fields.

For the CPP backend, @utf8InCpp String in AIDL maps to std::string, whereas String without the annotation maps to android::String16 where UTF16 is used.

Note that the existence of the utf8InCpp annotation doesn't change the way strings are transmitted over the wire. Strings are always transmitted as UTF16 over the wire. A utf8InCpp annotated string is converted to UTF16 before it is transmitted. When a string is received, it is converted from UTF16 to UTF8 if it was annotated as utf8InCpp.

VintfStability
VintfStability declares that a user-defined type (interface, parcelable, and enum) can be used across the system and vendor domains. See AIDL for HALs for more about system-vendor interoperability.

The annotation doesn't change the signature of the type, but when it is set, the instance of the type is marked as stable so that it can travel across the vendor and system processes.

The annotation can only be attached to user-defined type declarations as shown here:

```
@VintfStability
interface IFoo {
    ....
}

@VintfStability
parcelable Data {
    ....
}

@VintfStability
enum Type {
    ....
}
```

When a type is annotated with VintfStability, any other type that is referenced in the type should also be annotated as such. In the following example, Data and IBar should both be annotated with VintfStability.

```
@VintfStability
interface IFoo {
    void doSomething(in IBar b); // references IBar
    void doAnother(in Data d); // references Data
}

@VintfStability // required
interface IBar {...}
```

@VintfStability // required
parcelable Data {...}
In addition, the AIDL files defining types annotated with VintfStability can only be built using the aidl_interface Soong module type, with the stability property set to "vintf".

```
aidl_interface {
    name: "my_interface",
    srcs: [...],
    stability: "vintf",
}
```

UnsupportedAppUsage
The UnsupportedAppUsage annotation denotes that the annotated AIDL type is part of the non-SDK interface that has been accessible for legacy apps. See Restrictions on non-SDK interfaces for more information about the hidden APIs.

The UnsupportedAppUsage annotation doesn't affect the behavior of the generated code. The annotation only annotates the generated Java class with the Java annotation of the same name.

```
// in AIDL
@UnsupportedAppUsage
interface IFoo {...}

// in Java
@android.compat.annotation.UnsupportedAppUsage
public interface IFoo {...}
```

This is a no-op for non-Java backends.

Backing
The Backing annotation specifies the storage type of an AIDL enum type.

```
@Backing(type="int")
enum Color { RED, BLUE, }
```

In the CPP backend, this emits a C++ enum class of type int32_t.

```
enum class Color : int32_t {
    RED = 0,
    BLUE = 1,
}
```

If the annotation is omitted, the type is assumed to be byte, which maps to int8_t for the CPP backend.

The type argument can be set only to the following integral types:

byte (8-bit wide)
int (32-bit wide)
long (64-bit wide)
NdkOnlyStableParcelable
NdkOnlyStableParcelable marks a parcelable declaration (not definition) as stable so that it can be referenced from other stable AIDL types. This is like JavaOnlyStableParcelable, but NdkOnlyStableParcelable marks a parcelable declaration as stable for the NDK backend instead of for Java.

To use this parcelable:

You must specify ndk_header.
You must have an NDK library specifying the parcelable and the library must be compiled into the library. For example, in the core build system on a cc_* module, use static_libs or shared_libs. For aidl_interface, add the library under additional_shared_libraries in Android.bp.
JavaOnlyStableParcelable
JavaOnlyStableParcelable marks a parcelable declaration (not definition) as stable so that it can be referenced from other stable AIDL types.

Stable AIDL requires that all user-defined types are stable. For parcelables, being stable requires that its fields are explicitly described in the AIDL source file.

```
parcelable Data { // Data is a structured parcelable.
```

```
  int x;
  int y;
}

parcelable AnotherData { // AnotherData is also a structured parcelable
  Data d; // OK, because Data is a structured parcelable
}
```

If the parcelable was unstructured (or just declared), then it can't be referenced.

```
parcelable Data; // Data is NOT a structured parcelable

parcelable AnotherData {
  Data d; // Error
}
```

JavaOnlyStableParcelable lets you to override the check when the parcelable you are referencing is already safely available as part of the Android SDK.

```
@JavaOnlyStableParcelable
parcelable Data;

parcelable AnotherData {
  Data d; // OK
}
```

JavaDerive
JavaDerive automatically generates methods for parcelable types in the Java backend.

```
@JavaDerive(equals = true, toString = true)
parcelable Data {
 int number;
 String str;
}
```

The annotation requires additional parameters to control what to generate. The supported parameters are:

equals=true generates equals and hashCode methods.
toString=true generates toString method that prints the name of the type and fields. For example: Data{number: 42, str: foo}
JavaDefault
JavaDefault, added in Android 13, controls whether the default implementation versioning support is generated (for setDefaultImpl). This support is no longer generated by default in order to save space.

JavaPassthrough
JavaPassthrough lets the generated Java API be annotated with an arbitrary Java annotation.

The following annotations in AIDL

```
@JavaPassthrough(annotation="@android.annotation.Alice")
@JavaPassthrough(annotation="@com.android.Alice(arg=com.android.Alice.Value.A)")
```

become

```
@android.annotation.Alice
@com.android.Alice(arg=com.android.Alice.Value.A)
```

in the generated Java code.

The value of the annotation parameter is directly emitted. The AIDL compiler doesn't look into the value of the parameter. If there is any Java-level syntax error, it won't be caught by the AIDL compiler but by the Java compiler.

This annotation can be attached to any AIDL entity. This annotation is a no-op for non-Java backends.

RustDerive
RustDerive automatically implements traits for generated Rust types.

The annotation requires additional parameters to control what to generate. The supported parameters are:

Copy=true
Clone=true
Ord=true
PartialOrd=true
Eq=true
PartialEq=true
Hash=true
For explanations of these traits, see https://doc.rust-lang.org.

FixedSize
FixedSize marks a structured parcelable as fixed size. Once marked, the parcelable won't be allowed to have new fields added to it. All fields of the parcelable must also be fixed sized types, including primitive types, enums, fixed-size arrays, and other parcelables marked with FixedSize.

This doesn't provide any guarantee across different bitnesses and shouldn't be relied on for mixed-bitness communication.

Descriptor
Descriptor forcibly specifies the interface descriptor of an interface.

```
package android.foo;

@Descriptor(value="android.bar.IWorld")
interface IHello {...}
```

The descriptor of this interface is android.bar.IWorld. If the Descriptor annotation is missing, the descriptor would be android.foo.IHello.

This is useful for renaming an already published interface. Making the descriptor of the renamed interface the same as the descriptor of the interface before the renaming allows the two interfaces to talk to each other.

@hide in comments
The AIDL compiler recognizes @hide in comments and passes it through to Java output for metalava to pickup. This comment ensures that the Android build system knows that AIDL APIs are not SDK APIs.

@deprecated in comments
The AIDL compiler recognizes @deprecated in comments as a tag to identify an AIDL entity that should no longer be used.

```
interface IFoo {
  /** @deprecated use bar() instead */
  void foo();
  void bar();
}
```

Each backend marks deprecated entities with a backend-specific annotation or attribute so that the client code is warned if it refers the deprecated entities. For example, the @Deprecated annotation and the @deprecated tag are attached to the Java generated code.

Here's a **clear and simple explanation** of **AIDL annotations** with **examples**, covering:

1. ✅ What each annotation does
2. ✅ Where you can use it
3. ✅ Realistic AIDL code examples
4. ✅ How it behaves across **Java / CPP / NDK / Rust**

📘 What Are AIDL Annotations?

AIDL annotations are like **tags** that provide **extra instructions** to the AIDL compiler.

They:

- Modify how code is **generated** for each backend (Java, NDK, CPP, Rust)
- Control behavior like **nullability**, **stability**, **string format**, **interface name**, etc.
- Are **predefined only** — you can't create custom annotations

---

🔠 Annotation Syntax

```aidl
@AnnotationName(arguments) AidlEntity
```

Example:

```aidl
@nullable Data method();
```

🔍 Key Annotations Explained with Examples

1. `@nullable` (Android 7+)

✅ Means the field or argument can be **null**.
🚫 Can't be used on primitive types like `int`, `boolean`.

```aidl
interface IFoo {
    @nullable Data getData();          // nullable return
    void setData(in @nullable Data d); // nullable parameter
}

parcelable Data {
    @nullable String name;           // nullable field
}
```

🔁 Mapped as:

- Java: all objects are nullable by default (noop)
- NDK: `std::optional<T>`
- CPP (Android 11): `std::unique_ptr<T>`
- CPP (Android 12+): `std::optional<T>`
- Rust: `Option<T>`

---

2. `@utf8InCpp` (Android 7+)

✅ Makes a `String` be treated as UTF-8 in **CPP backend**.

```aidl
@utf8InCpp String getName();
```

🔁 Maps to:

- CPP: `std::string` (instead of `android::String16`)
- NDK: always UTF-8 already (noop)
- Java: always UTF-16 (noop)

---

3. `@VintfStability` (Android 11+)

✅ Declares that this type is **system-vendor stable** and can cross boundaries (e.g., used in AIDL HALs).
🔒 Required when your AIDL is in a HAL that spans system/vendor partitions.

```aidl
@VintfStability
interface IFoo {
    void doSomething(in Bar b);
}

@VintfStability
```

```
interface Bar {}

@VintfStability
parcelable Data {}
```

📦 Must use in a **`aidl_interface`** Soong module with **`stability: "vintf"`**.

---

4. `@Backing(type="int")` (Android 11+)

✅ Controls the **underlying type** of an enum.

```aidl
@Backing(type="int")
enum Mode {
    AUTO = 0,
    MANUAL = 1,
}
```

🔁 Maps to:

- CPP: **`enum class Mode : int32_t`**
- If not given, default is **`byte`** (8-bit)

---

5. `@JavaOnlyStableParcelable` / `@NdkOnlyStableParcelable`

✅ Marks **parcelables** that are already defined in Android SDK as **stable**, so you can safely reference them.

```aidl
@JavaOnlyStableParcelable
parcelable android.graphics.Rect;

parcelable WindowInfo {
    android.graphics.Rect bounds; // OK now
}
```

---

6. `@JavaDerive`

✅ Auto-generates useful Java methods for parcelables.

```aidl
@JavaDerive(equals = true, toString = true)
parcelable Point {
    int x;
    int y;
}
```

🧾 Generates:

- **`equals()`**, **`hashCode()`**, **`toString()`** in Java backend

---

7. `@JavaPassthrough`

✅ Injects raw annotations into generated Java.

```aidl
@JavaPassthrough(annotation="@android.annotation.SystemApi")
interface IFoo {}
```

📌 In generated Java:

```java
@android.annotation.SystemApi
public interface IFoo { ... }
```

---

8. `@FixedSize` (Android 12+)

✅ Marks a parcelable as **fixed-size**, meaning:

- All fields must also be fixed-size (int, float, enums, fixed-size arrays)
- Can't be extended later

```
aidl

@FixedSize
parcelable Header {
    int magic;
    int size;
}
```

Useful for binary protocols but ⚠️ not portable across 32/64-bit.

---

9. `@Descriptor` (Android 12+)

✅ Renames the interface **descriptor** (the string used in Binder registry).

```
aidl

package com.vendor.foo;

@Descriptor(value="com.vendor.legacy.IFoo")
interface IFoo {}
```

🧾 Now appears as `com.vendor.legacy.IFoo`, not `com.vendor.foo.IFoo`.

---

10. `@hide` and `@deprecated` (in comments)

✅ Used to hide or deprecate methods.

```
aidl

interface IFoo {
    /** @hide */
    void internalCall();

    /** @deprecated use bar() instead */
    void foo();

    void bar();
}
```

🔁 The compiler passes these to:

- Java: `@Deprecated`, metalava tags
- NDK/CPP: adds warnings via attributes

🧠 Summary Table

Annotation	Purpose	Example
`@nullable`	Allows null	`@nullable Data get()`
`@utf8InCpp`	Use UTF-8 in CPP	`@utf8InCpp String name`
`@VintfStability`	HAL system ↔ vendor support	`@VintfStability interface IHal`
`@Backing`	Enum storage size	`@Backing(type="int") enum Mode`
`@JavaOnlyStableParcelable`	Mark SDK parcelables as stable	`@JavaOnlyStableParcelable parcelable Rect`
`@JavaDerive`	Auto-generate Java methods	`@JavaDerive(equals = true)`
`@JavaPassthrough`	Add Java annotations	`@JavaPassthrough(annotation="@SystemApi")`
`@FixedSize`	Prevent structure change	`@FixedSize parcelable Header`
`@Descriptor`	Rename interface	`@Descriptor("com.legacy.IFoo")`
`@hide`, `@deprecated`	Hide from SDK / warn users	`/** @deprecated */`

Let me know if you want a **working AIDL example project** with annotations across Java/NDK/CPP, or details on **parcelable extensions or versioning**.

16. Fast Message Queue with AIDL

bookmark_border
As of Android 12, Fast Message Queue can be used with AIDL interfaces using the NDK backend. This allows processes to communicate without the overhead and restrictions of binder transactions after some short setup. Using Stable AIDL allows communication between system and vendor processes.

Supported payload types
@FixedSize AIDL parcelable types
AIDL enum types
AIDL integral types
The messages sent between processes in the shared memory message queue must have the same memory layout across process boundaries and cannot contain pointers. Attempting to create an AidlMessageQueue with a type that isn't supported will cause a compilation error.

Supported queue types
The same queue types from HIDL, often called flavors, are supported with AIDL. These are used as template arguments for the queues and descriptors.

HIDL Types AIDL Types
android::hardware::kSynchronizedReadWrite android.hardware.common.fmq.SynchronizedReadWrite
android::hardware::kUnsynchronizedWrite android.hardware.common.fmq.UnsynchronizedWrite
How to use
Define the AIDL interface that will pass the MQDescriptor to the other process. MQDescriptor can be used anywhere a parcelable can be.

The required template arguments for MQDescriptor are payload type and queue flavor.

```
import android.hardware.common.fmq.MQDescriptor
import android.hardware.common.fmq.SynchronizedReadWrite

void getQueue(out MQDescriptor<int, SynchronizedReadWrite> mqDesc);
```
The process of setting up each side of the message queue is nearly identical to the process using HIDL, just using the AIDL types.

```
#include <fmq/AidlMessageQueue.h>
...
using ::android::AidlMessageQueue;
using ::aidl::android::hardware::common::fmq::MQDescriptor;
using ::aidl::android::hardware::common::fmq::SynchronizedReadWrite;
...
ndk::ScopedAStatus MyInterface::getQueue(MQDescriptor<int32_t, SynchronizedReadWrite>* mqDesc) {
    *mqDesc = mFmqSynchronized->dupeDesc();
    return ndk::ScopedAStatus::ok();
}
...
// Create the first side of the queue before servicing getQueue() in this example
mFmqSynchronized =
  new AidlMessageQueue<int32_t, SynchronizedReadWrite>(kNumElementsInQueue);
```
The receiving process will create the other side of the queue with the descriptor received from the AIDL interface.

```
MQDescriptor<int32_t, SynchronizedReadWrite> desc;
auto ret = service->getQueue(true, &desc);
if (!ret.isOk()) {
   ...
}
// By default the constructor will reset the read and write pointers of the queue.
// Add a second false argument to avoid resetting the pointers.
mQueue = new (std::nothrow) AidlMessageQueue<int32_t, SynchronizedReadWrite>(desc);
if (!mQueue->isValid()) {
   ...
}
```
Using the AidlMessageQueue after setup is the same as the HIDL MessageQueue. All of the APIs described at Using the MessageQueue are fully

supported with AidlMessageQueue with one exception:

const MQDescriptor<T, flavor>* getDesc() is replaced by MQDescriptor<T, U> dupeDesc() which returns the AIDL MQDescriptor.

Here's a **detailed and beginner-friendly explanation** of using **Fast Message Queue (FMQ) with AIDL (NDK backend)** — introduced in **Android 12**.

🧠 What is FMQ?

Fast Message Queue (FMQ) is a **zero-copy, shared-memory** mechanism for **fast interprocess communication** (IPC) between system and vendor.

🔄 Unlike binder IPC (which copies data), FMQ shares a memory buffer between processes.
📦 Only supports **fixed-size types** (like `int32_t`, enums, or @FixedSize parcelables) — no pointers or heap objects.

✅ When to Use FMQ in AIDL?

Use FMQ when you need:

- High-performance IPC (like sensor data streaming)
- Repetitive small messages (e.g., audio samples, event signals)
- Deterministic latency and **low binder overhead**

🚧 AIDL Requirements for FMQ

To use FMQ in AIDL:

- Use **NDK backend** only (Java not supported)
- Only supported types:
  - `@FixedSize` parcelables
  - AIDL enums
  - Integral types (`int32_t`, `uint8_t`, etc.)
- Use `MQDescriptor<T, Flavor>` to pass queue handles
- Use `AidlMessageQueue<T, Flavor>` C++ API to access the queue

🔁 Supported Queue Flavors (Modes)

Flavor name	Description
`SynchronizedReadWrite`	Locking between reader/writer
`UnsynchronizedWrite`	Assumes only one writer thread, no locks

Defined in:

```aidl
import android.hardware.common.fmq.SynchronizedReadWrite;
import android.hardware.common.fmq.UnsynchronizedWrite;
```

🧩 FMQ + AIDL Interface Example

1. AIDL Interface

```aidl
// IMyService.aidl
import android.hardware.common.fmq.MQDescriptor;
import android.hardware.common.fmq.SynchronizedReadWrite;

interface IMyService {
```

```
    void getQueue(out MQDescriptor<int, SynchronizedReadWrite> desc);
}
```

---

2. Server-side C++ (NDK)

Header

```cpp
#include <aidl/package/name/IMyService.h>
#include <fmq/AidlMessageQueue.h>

using aidl::package::name::IMyService;
using aidl::android::hardware::common::fmq::SynchronizedReadWrite;
using aidl::android::hardware::common::fmq::MQDescriptor;
using android::AidlMessageQueue;
```

Class

```cpp
class MyService : public IMyService {
private:
    std::unique_ptr<AidlMessageQueue<int32_t, SynchronizedReadWrite>> mQueue;

public:
    MyService() {
        // Allocate the message queue
        mQueue = std::make_unique<AidlMessageQueue<int32_t, SynchronizedReadWrite>>(128);
    }

    ndk::ScopedAStatus getQueue(MQDescriptor<int32_t, SynchronizedReadWrite>* desc) override {
        *desc = mQueue->dupeDesc();  // pass the queue descriptor to client
        return ndk::ScopedAStatus::ok();
    }

    // Optionally: producer side
    void pushData(int32_t value) {
        mQueue->writeBlocking(&value, 1, 1000000); // 1ms timeout
    }
};
```

---

3. Client-side C++ (NDK)

```cpp
using aidl::android::hardware::common::fmq::MQDescriptor;
using aidl::android::hardware::common::fmq::SynchronizedReadWrite;

std::shared_ptr<IMyService> service = IMyService::fromBinder(binder);

MQDescriptor<int32_t, SynchronizedReadWrite> desc;
if (!service->getQueue(&desc).isOk()) {
    // handle error
}

// Create consumer-side queue using descriptor
auto mQueue = std::make_unique<AidlMessageQueue<int32_t, SynchronizedReadWrite>>(desc);

if (!mQueue->isValid()) {
    // handle invalid queue
}

// Read data
int32_t value;
bool success = mQueue->readBlocking(&value, 1, 1000000); // 1ms timeout
```

🧵 Summary of Key Methods

Method	Description
`write(data, count)`	Non-blocking write
`writeBlocking(data, count, timeout)`	Blocking write
`read(buffer, count)`	Non-blocking read
`readBlocking(buffer, count, timeout)`	Blocking read

Method	Description
`dupeDesc()`	Get AIDL-compatible `MQDescriptor` for passing
`isValid()`	Check if FMQ is set up properly

📌 Important Notes

- Both client and server must **use the exact same template types**:
  `AidlMessageQueue<int32_t, SynchronizedReadWrite>` on both sides.
- Cannot use non-fixed-size types like:
  - `String`
  - `List<T>`
  - non-@FixedSize parcelables
  - interfaces or binders
- No heap or dynamic memory structures allowed in FMQ payloads.

✅ Use Case Example

- 🎧 **Audio HAL**: Send audio samples quickly to DSP
- 🧭 **Sensor HAL**: Stream accelerometer or gyroscope data
- 📷 **Camera HAL**: Transmit small control/status packets rapidly

Would you like a complete **NDK-based AIDL + FMQ Android project structure** to try this in practice?

17. AIDL fuzzing

bookmark_border
The fuzzer behaves as a client for the remote service by importing or invoking it through the generated stub:

Using C++ API:

```
#include <fuzzbinder/libbinder_ndk_driver.h>
#include <fuzzer/FuzzedDataProvider.h>

#include <android-base/logging.h>
#include <android/binder_interface_utils.h>

using android::fuzzService;
using ndk::SharedRefBase;

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
   auto binder = ndk::SharedRefBase::make<MyService>(...);

   fuzzService(binder->asBinder().get(), FuzzedDataProvider(data, size));

   return 0;
}
```

Using Rust API:

```
#![allow(missing_docs)]
#![no_main]
#[macro_use]
extern crate libfuzzer_sys;

use binder::{self, BinderFeatures, Interface};
use binder_random_parcel_rs::fuzz_service;

fuzz_target!(|data: &[u8]| {
   let service = BnTestService::new_binder(MyService, BinderFeatures::default());
   fuzz_service(&mut service.as_binder(), data);
});
```

Framework to fuzz AIDL services
As shown in the example above, fuzzService is called in the fuzzer and takes in an IBinder (Service) and dataProvider as input parameters. It first initializes a random Parcel object using the data provider and call the transact method on the remote service by using the input parcel, and finally get the reply into a reply parcel.

Build and run fuzzers
Fuzzers are built with coverage by default.

The following sanitizers are recommended to discover memory issues. hwaddress sanitizers only run on arm architecture:

```
SANITIZE_HOST=address SANITIZE_TARGET=hwaddress
```

When running with libFuzzer, a corpus, which is a directory, may be specified in the Android.bp file, and you can pass this directory to the fuzzer. Some fuzzers also specify a dictionary: in their Android.bp file, and you can pass this to libFuzzer with -dict path/to/dict. For more options, see the official libFuzzer documentation.

To run fuzzers on device, run adb sync data and then adb shell data/fuzz/arch/name/name. To run fuzzers on host, run $ANDROID_HOST_OUT/fuzz/arch/name/name.

Recommend fuzzers for new or existing services
The build system checks whether every AOSP binder service has a fuzzer entry in service fuzzer bindings. The Fuzzer binding test checks that every service in service_contexts has a fuzzer. If a fuzzer or exception isn't found for a new service, there's a build error.

An automatic C++ service fuzzer can be written by adding the following (Java and Rust fuzzers are not yet supported):

A cc_fuzz entry in Android.bp to define the fuzzer module. The cc_default module service_fuzzer_defaults has dependencies required for fuzzService.
Service-specific dependencies should be added as a library or as sources.
A main file that constructs your service and calls fuzzService
For detailed instructions on using cc_fuzz, see the Fuzzing with libFuzzer documentation. To resolve build error, update bindings with the new service and fuzzer names. For Java or Rust services, the fuzzer list can be empty.

Here's a **detailed, step-by-step explanation of AIDL service fuzzing** in Android, particularly for **NDK-based AIDL services** using the `fuzzbinder` framework.

🧠 What is AIDL Fuzzing?

Fuzzing is a security and stability testing method where a service is bombarded with **random, malformed, or unexpected inputs** to catch:

- **Crashes**
- **Memory corruption**
- **Unexpected behavior**

For AIDL, fuzzing helps ensure **robustness of binder services** by simulating a client sending corrupted `Parcel` data.

🔁 High-Level Flow

1. Fuzzer creates a fake client binder (`IBinder`).
2. Random bytes are interpreted as `Parcel` data using `FuzzedDataProvider`.
3. These bytes are sent via `transact()` to the AIDL service.
4. The fuzzer observes if the service:
    - Crashes
    - Hangs
    - Misbehaves

🛠️ Tools Used

Tool/Library	Purpose
`fuzzbinder`	Binder-specific fuzzing APIs
`FuzzedDataProvider`	Feeds random input data
`libFuzzer`	Executes fuzzing with instrumentation
`clang` + Sanitizers	Helps detect memory issues (ASan, HWASan)

🧪 Example: C++ AIDL Service Fuzzing

1. Fuzzer Main File (`my_service_fuzzer.cpp`)

```cpp
#include <fuzzbinder/libbinder_ndk_driver.h>
#include <fuzzer/FuzzedDataProvider.h>
#include <android/binder_interface_utils.h>
#include "aidl/com/example/MyService.h" // AIDL service
#include "MyServiceImpl.h"              // Your implementation

using android::fuzzService;
using ndk::SharedRefBase;
using com::example::MyService;

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    // Construct service instance (stub)
    auto service = ndk::SharedRefBase::make<MyServiceImpl>();

    // Fuzz binder transactions on this service
    fuzzService(service->asBinder().get(), FuzzedDataProvider(data, size));
```

```
    return 0;
}
```

2. Android.bp for Fuzzer

```bp
cc_fuzz {
    name: "my_service_fuzzer",
    defaults: ["service_fuzzer_defaults"],

    srcs: ["my_service_fuzzer.cpp"],

    shared_libs: [
        "libbinder_ndk",
        "libbase",
        "liblog",
        "libutils",
    ],
    static_libs: [
        "libfuzzbinder",
        "libFuzzedDataProvider",
        "libMyServiceStub",  // Stub generated from AIDL
        "libMyServiceImpl",  // Your implementation
    ],
    fuzz_config: {
        cc: [
            "my-service@aosp.com",
        ],
    },
}
```

🧪 Optional Rust Fuzzer Example

```rust
#![no_main]
#[macro_use]
extern crate libfuzzer_sys;

use binder::{self, Interface, BinderFeatures};
use binder_random_parcel_rs::fuzz_service;

fuzz_target!(|data: &[u8]| {
    let service = BnTestService::new_binder(MyService, BinderFeatures::default());
    fuzz_service(&mut service.as_binder(), data);
});
```

> 🟨 Rust fuzzing is available but currently **optional**. Java fuzzing is not supported yet.

🧪 Build + Run

Build (Host or Target)

```sh
m my_service_fuzzer
```

Run on Device

```sh
adb sync data
adb shell /data/fuzz/arm64/my_service_fuzzer/my_service_fuzzer
```

Run on Host

```sh
$ANDROID_HOST_OUT/fuzz/x86_64/my_service_fuzzer/my_service_fuzzer
```

🧪 Use Sanitizers

Enable sanitizers to detect memory issues:

```sh
SANITIZE_HOST=address SANITIZE_TARGET=hwaddress m my_service_fuzzer
```

> Note: **`hwaddress`** sanitizer works only on ARM64 targets.

✅ Integrating Fuzzer into the Build

If your service is added to **`service_contexts`**, you **must** create a fuzzer or opt out using an exception list. Otherwise, the build will **fail**.

To integrate:

- Add fuzzer into **`cc_fuzz`**
- Ensure your service is part of the **`service_fuzzer_bindings.txt`**

📌 Recommendations

If your service is...	Then...
A new binder service	Write a fuzzer and register it
A Java service	OK to skip (not supported)
A Rust AIDL service	Optional (early-stage support)
Using fixed parcelables	Consider adding FMQ or parcel fuzzer too

---

🧩 Advanced: Fuzz Multiple Methods

- The fuzzer will automatically target **all methods** exposed via **`transact()`**, including:
  - All AIDL methods
  - Parcelables
  - Exceptions
- If a crash or memory violation occurs during fuzzing, it's considered a bug or a security flaw.

Would you like a **complete working fuzzing example project** or help writing a **MyServiceImpl stub class**?

18. yes but explain each code

Sure! Let's break down the C++ AIDL fuzzing example step-by-step and explain what each line does:

✅ Full Code Snippet:

```cpp
#include <fuzzbinder/libbinder_ndk_driver.h>
#include <fuzzer/FuzzedDataProvider.h>

#include <android-base/logging.h>
#include <android/binder_interface_utils.h>

using android::fuzzService;
using ndk::SharedRefBase;

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    auto binder = ndk::SharedRefBase::make<MyService>(...);

    fuzzService(binder->asBinder().get(), FuzzedDataProvider(data, size));

    return 0;
}
```

🔍 Line-by-Line Explanation:

✅ `#include <fuzzbinder/libbinder_ndk_driver.h>`

- **Purpose:** Includes the header for the `fuzzService()` helper function.
- This is the core helper provided by Android for fuzzing NDK-based AIDL services.
- It sets up the environment, handles fuzz data parsing, and performs Binder transactions.

✅ `#include <fuzzer/FuzzedDataProvider.h>`

- **Purpose:** Provides the `FuzzedDataProvider` class, which feeds fuzzed input to the target.
- **What it does:** Interprets the byte stream (`data`, `size`) in a structured way so you can extract strings, ints, booleans, etc., from fuzz input safely.

✅ `#include <android-base/logging.h>`

- **Purpose:** Android logging support (optional in some fuzzers).
- Allows you to log debug/error info using `LOG(INFO)`, `LOG(ERROR)`, etc.

✅ `#include <android/binder_interface_utils.h>`

- **Purpose:** Includes utilities for dealing with AIDL binder interfaces.
- Includes `SharedRefBase`, interface stubs, and other AIDL runtime support functions for NDK backend.

✅ `using android::fuzzService;`

- **Purpose:** Brings `fuzzService` into the current namespace for easier use.
- `fuzzService` is the actual function that performs the fuzz test by simulating transactions on your service.

✅ `using ndk::SharedRefBase;`

- **Purpose:** Brings the `SharedRefBase` type into current scope.
- A base class to implement `IBinder` interfaces with automatic reference counting.

🧠 Fuzzer Entry Point

```cpp
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size)
```

- **This is the main entry point for libFuzzer.**
- It's called repeatedly by the fuzzer engine with different mutated inputs.
- **`data`**: pointer to the fuzzed input data.
- **`size`**: length of the fuzzed data.

🚀 Inside the Fuzzing Function

```cpp
auto binder = ndk::SharedRefBase::make<MyService>(...);
```

- **Instantiates your AIDL service implementation** (`MyService` is your actual server class).
- `SharedRefBase::make` wraps it in a smart pointer with ref-counting.
- This is necessary because `fuzzService()` expects a full `IBinder` object, not just a raw service instance.

✅ You would normally replace `...` with any constructor arguments your service needs.

```cpp
fuzzService(binder->asBinder().get(), FuzzedDataProvider(data, size));
```

- **Core fuzzing step.**
- `binder->asBinder().get()`: gets the `IBinder*` interface for the AIDL service.
- `FuzzedDataProvider(data, size)`: feeds the input data to generate a fake Parcel transaction.
- `fuzzService(...)`: simulates a client calling your AIDL service using a random Parcel as input.

It:

1. Creates a random input `Parcel`.
2. Sends it to your service as a transaction.
3. Collects the reply and validates memory correctness.

```cpp
return 0;
```

- Standard way to signal success (no crash).
- libFuzzer checks this to know whether the fuzz test crashed.

💡 Summary

Component	Role
`LLVMFuzzerTestOneInput`	Entry point for libFuzzer. Feeds mutated data into service.
`SharedRefBase::make`	Creates an instance of the service under test.
`fuzzService`	Sends a random Parcel to the service and watches for crashes.
`FuzzedDataProvider`	Interprets fuzz data in structured ways (int, string, bool, etc.)
`IBinder` and `asBinder`	Core binder interface used to perform transactions.

Would you also like me to explain the Rust version line-by-line or go deeper into how fuzzing discovers issues (like OOB, UAF, etc.)?

19. complete working fuzzing example project

Here's a complete working AIDL fuzzing example project in C++ using **`fuzzService`**, targeting an AIDL service implemented with the NDK backend. This includes:

📁 Directory Structure:

```bash
aidl_fuzz_example/
├── aidl/
│   └── com/example/ITestService.aidl
├── service/
│   ├── Android.bp
│   ├── MyService.cpp
│   └── MyService.h
├── fuzz/
│   ├── Android.bp
│   └── ITestService_fuzzer.cpp
```

🔧 Step-by-step Breakdown:

1. ✅ AIDL Interface – `ITestService.aidl`

aidl/com/example/ITestService.aidl

```aidl
package com.example;

interface ITestService {
    void doSomething(int value);
}
```

2. ✅ C++ Service Implementation

service/MyService.h

```cpp
#pragma once

#include <aidl/com/example/BnTestService.h>

class MyService : public aidl::com::example::BnTestService {
  public:
    ndk::ScopedAStatus doSomething(int32_t value) override;
};
```

service/MyService.cpp

```cpp
#include "MyService.h"
#include <android-base/logging.h>

ndk::ScopedAStatus MyService::doSomething(int32_t value) {
    LOG(INFO) << "Received value in doSomething: " << value;
    return ndk::ScopedAStatus::ok();
}
```

3. ✅ Fuzzer Source

fuzz/ITestService_fuzzer.cpp

```cpp
#include <fuzzbinder/libbinder_ndk_driver.h>
#include <fuzzer/FuzzedDataProvider.h>
#include <android/binder_interface_utils.h>

#include "MyService.h"

using android::fuzzService;
using ndk::SharedRefBase;

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    // Instantiate the service
    auto service = SharedRefBase::make<MyService>();

    // Call the fuzzService which will:
    // 1. Use the data to fill a Parcel
    // 2. Call the transact() function on the binder
    // 3. Receive the reply
    fuzzService(service->asBinder().get(), FuzzedDataProvider(data, size));

    return 0;
}
```

---

4. ✅ Android.bp Files

service/Android.bp

```bp
cc_library_shared {
    name: "libmyservice",
    srcs: [
        "MyService.cpp",
    ],
    shared_libs: [
        "libbinder_ndk",
        "liblog",
    ],
    export_include_dirs: ["."],
}
```

fuzz/Android.bp

```bp
cc_fuzz {
    name: "itestservice_fuzzer",
    srcs: ["ITestService_fuzzer.cpp"],
    shared_libs: [
        "libbinder_ndk",
        "liblog",
        "libmyservice",
    ],
    static_libs: [
        "libfuzzbinder_ndk_driver",
    ],
    fuzz_config: {
        cc: [
            "example-team@example.com",
        ],
    },
}
```

---

5. ✅ AIDL Interface Declaration

If you're using Soong:

```bp
aidl_interface {
    name: "com.example.itestservice",
    srcs: ["aidl/com/example/ITestService.aidl"],
    backend: {
        ndk: {
            enabled: true,
        },
    },
    versions: ["1"],
    stability: "vintf",
}
```

✅ Building

In your AOSP repo:

```sh
source build/envsetup.sh
lunch <your_target>
m
```

✅ Running

Push fuzzer to device:

```sh
adb push out/fuzz_target/<arch>/itestservice_fuzzer /data/local/tmp/
adb shell /data/local/tmp/itestservice_fuzzer
```

Let me know if you want:

- Java or Rust version instead
- AIDL with multiple methods / complex data
- Corpus or dictionary generation
- Crash triage explanation

Want me to zip this example as source code or convert it to a GitHub repo format for easy cloning?

20. ↪
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) { // Instantiate the service auto service = SharedRefBase::make<MyService>(); // Call the fuzzService which will: // 1. Use the data to fill a Parcel // 2. Call the transact() function on the binder // 3. Receive the reply fuzzService(service->asBinder().get(), FuzzedDataProvider(data, size)); return 0; }
explain

Let's go through the `LLVMFuzzerTestOneInput()` function **line by line**, with **detailed explanation** of each part in the context of AIDL fuzzing:

🔧 Function Definition

```cpp
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size)
```

- **`extern "C"`**:
  Tells the compiler **not to mangle the function name**, so that **libFuzzer** can find it at runtime.
- **`LLVMFuzzerTestOneInput`**:
  This is the **entry point** for a **libFuzzer** fuzz test. libFuzzer repeatedly calls this function with random input.
- **`const uint8_t* data, size_t size`**:
  This is the **random byte array** and its length that libFuzzer will generate. You must use this to feed malformed or interesting inputs into your service under test.

🛠️ Instantiate the Service

```cpp
auto service = SharedRefBase::make<MyService>();
```

- **`SharedRefBase::make<T>()`**:
  Allocates and returns a **shared reference-counted binder interface object** (`MyService`) using the Android NDK Binder infrastructure.
- `MyService` is your service class that implements the `ITestService` AIDL interface.
  You're creating a local in-process service here.

🔍 Perform the Fuzzing

```cpp
fuzzService(service->asBinder().get(), FuzzedDataProvider(data, size));
```

🔄 Breaking it Down:

`service->asBinder().get()`

- Converts the `MyService` object to a raw `IBinder*`.
- `asBinder()` returns a smart pointer (`sp<IBinder>`), and `.get()` returns the raw pointer.
- This is the **binder endpoint** used for communication.

`FuzzedDataProvider(data, size)`

- Converts the raw fuzz input into a `FuzzedDataProvider` object.
- This lets you **incrementally consume** chunks of the fuzz input (e.g., 4 bytes as `int`, next 6 as string, etc.).

`fuzzService(...)`

This function is the **core fuzzing driver** provided by Android's `libbinder_ndk_driver`.

It works by:

1. **Generating a random** `Parcel` using the fuzzed data.
2. **Calling** `transact()` on the `IBinder` with the random `Parcel` as input.
3. **Waiting for the service** (your `MyService`) to handle the call and reply.
4. **Receiving the output** in a reply `Parcel`.
5. Checking for crashes, memory corruptions, or hangs.

This simulates how a **malicious client** might send corrupted or invalid data to your service.

✅ Return from Fuzzer Function

```cpp
return 0;
```

- libFuzzer expects `int` return value.
- Return `0` means "no crash or bug found" in this iteration.

🧠 Summary

The entire function sets up and executes one fuzzing iteration:

Step	Action
1	Accepts fuzz input from libFuzzer
2	Wraps it as `FuzzedDataProvider`
3	Creates a real implementation of your AIDL service
4	Invokes `fuzzService()` to send random Parcel to service
5	Service receives, executes, and replies
6	Any crash/assertion/memory issue will be caught by fuzzer

📌 Why it's Powerful

This framework **fuzzes the actual Binder IPC code paths**, including:

- Parcel serialization
- Binder transaction logic
- Your AIDL method logic
- Type checking and error handling

If you have any of the following issues, this will catch them:

- Incorrect deserialization
- Memory corruption
- Logic errors (e.g., divide-by-zero)
- Buffer overflows in `doSomething`

Let me know if you want the **service side method** (`doSomething`) fuzzed deeper (e.g. via extra data parsing), or if you want to hook into logs or assertions.

21. yes

Great! Let's break down the complete working AIDL fuzzing example step-by-step so you understand how it works.

✅ Goal:

Fuzz a native (NDK) AIDL service using **`libFuzzer`** and **`fuzzService()`** to check for unexpected behavior, crashes, or vulnerabilities.

📁 Project Structure:

```
my_aidl_fuzzer/
├── Android.bp
├── IMyService.aidl
├── MyService.cpp
├── MyService.h
└── fuzz.cpp
```

🧩 1. `IMyService.aidl`

```aidl
package com.example.fuzz;

interface IMyService {
    int doSomething(int x);
}
```

➡️ This defines a simple AIDL service interface with a single method that takes an **`int`** and returns an **`int`**.

🧩 2. `MyService.h`

```cpp
#pragma once

#include "aidl/com/example/fuzz/BnMyService.h"

class MyService : public aidl::com::example::fuzz::BnMyService {
  public:
    ndk::ScopedAStatus doSomething(int32_t x, int32_t* _aidl_return) override;
};
```

➡️ This defines a C++ class that implements the **`IMyService`** AIDL interface.

🧩 3. `MyService.cpp`

```cpp
#include "MyService.h"
#include <android-base/logging.h>

ndk::ScopedAStatus MyService::doSomething(int32_t x, int32_t* _aidl_return) {
    LOG(INFO) << "doSomething called with x=" << x;
    *_aidl_return = x * 2;
    return ndk::ScopedAStatus::ok();
}
```

➡️ Implements the method by simply doubling the input. The method is fuzzed later to catch bugs if the logic grows more complex.

🧩 4. `fuzz.cpp`

```cpp
#include <fuzzbinder/libbinder_ndk_driver.h>
#include <fuzzer/FuzzedDataProvider.h>

#include "MyService.h"

using android::fuzzService;
using aidl::com::example::fuzz::IMyService;

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    auto service = ndk::SharedRefBase::make<MyService>();
    fuzzService(service->asBinder().get(), FuzzedDataProvider(data, size));
    return 0;
}
```

🔍 What happens here:

- **`FuzzedDataProvider`** parses raw fuzzer input (**`data`**, **`size`**).
- **`fuzzService()`** crafts random **`Parcel`** calls using that data to fuzz all functions of **`MyService`**.

🧩 5. `Android.bp`

```bp
cc_fuzz {
    name: "my_service_fuzzer",
    srcs: ["fuzz.cpp", "MyService.cpp"],
    shared_libs: [
        "libbinder_ndk",
        "liblog",
        "libutils",
    ],
    static_libs: [
        "libfuzzbinder_ndk",
    ],
    defaults: ["service_fuzzer_defaults"],
    fuzz_config: {
        cc: ["me@example.com"],
    },
    include_dirs: [
        "aidl/com/example/fuzz",
    ],
}
```

➡️ This builds a **`cc_fuzz`** module named **`my_service_fuzzer`**.

🔄 Build & Run

```bash
# Build with fuzzing sanitizer
SANITIZE_TARGET=address m my_service_fuzzer

# Deploy to device
adb sync data
adb shell /data/fuzz/arm64/my_service_fuzzer/my_service_fuzzer
```

✅ Summary of Each Role

File	Role
`IMyService.aidl`	Defines the interface to be tested
`MyService.h/cpp`	Implements the service logic
`fuzz.cpp`	libFuzzer entry point to fuzz AIDL via **`fuzzService()`**
`Android.bp`	Compiles everything into a fuzzer binary

Would you like me to help you create this as a GitHub repo or zip archive too?

22. ↪
adb sync data adb shell /data/fuzz/arm64/my_service_fuzzer/my_service_fuzzer
?

Great question.

Let's break down this command:

```bash
adb sync data
adb shell /data/fuzz/arm64/my_service_fuzzer/my_service_fuzzer
```

🔹 `adb sync data`

This command:

- Pushes all files from your `out/data/` directory (on the host) to the `/data/` directory on the **device or emulator**.
- In this context, it ensures the **fuzzer binary** (and possibly its corpus/dictionary files) are copied to `/data/fuzz/...` on the device.

👉 Typically used after building the fuzzer with `m my_service_fuzzer`.

🔹 `adb shell /data/fuzz/arm64/my_service_fuzzer/my_service_fuzzer`

This command:

- Runs the **fuzzer binary** on the device.
- The full path depends on architecture and module name. Here:
  - `arm64` = architecture
  - `my_service_fuzzer` = name of your `cc_fuzz` module
  - `my_service_fuzzer` (final part) = actual binary name

🔁 Combined Flow:

1. **Build the fuzzer**:

   ```bash
   SANITIZE_TARGET=address m my_service_fuzzer
   ```

2. **Sync to device**:

   ```bash
   adb sync data
   ```

3. **Run the fuzzer**:

   ```bash
   adb shell /data/fuzz/arm64/my_service_fuzzer/my_service_fuzzer
   ```

You can also pass options to `libFuzzer`, e.g.:

```bash
adb shell /data/fuzz/arm64/my_service_fuzzer/my_service_fuzzer -runs=10000 -max_len=64
```

Let me know if you want help interpreting logs or setting up input corpus/dictionary.

23. AIDL style guide

bookmark_border
The best practices outlined here serve as a guide to developing AIDL interfaces effectively and with attention to flexibility of the interface, particularly when AIDL is used to define an API or interact with API surfaces.

AIDL can be used to define an API when apps need to interface with each other in a background process or need to interface with the system. For more information about developing programming interfaces in apps with AIDL, see Android Interface Definition Language (AIDL). For examples of AIDL in practice, see AIDL for HALs and Stable AIDL.

Versioning
Every backward-compatible snapshot of an AIDL API corresponds to a version. To take a snapshot, run m <module-name>-freeze-api. Whenever a client or server of the API is released (for example, in a Mainline train), you need to take a snapshot and make a new version. For system-to-vendor APIs, this should happen with the yearly platform revision.

For more details and information about the type of changes that are allowed, see Versioning interfaces.

Key Point: Understanding how interfaces can be modified is critical to maintaining the flexibility you need.
API design guidelines
General
1. Document everything
Document every method for its semantics, arguments, use of built-in exceptions, service specific exceptions, and return value.
Document every interface for its semantics.
Document the semantic meaning of enums and constants.
Document whatever might be unclear to an implementer.
Provide examples where relevant.
2. Casing
Use upper camel casing for types and lower camel casing for methods, fields and arguments. For example, MyParcelable for a parcelable type and anArgument for an argument. For acronyms, consider the acronym a word (NFC -> Nfc).

[-Wconst-name] Enum values and constants should be ENUM_VALUE and CONSTANT_NAME

Interfaces
1. Naming
[-Winterface-name] An interface name should start with I like IFoo.

2. Avoid big interface with id-based "objects"
Prefer subinterfaces when there are many calls related to a specific API. This provides the following benefits:

Makes client or server code easier to understand
Makes the lifecycle of objects simpler
Takes advantage of binders being unforgeable.
Not recommended: A single, large interface with ID-based objects

```
interface IManager {
  int getFooId();
  void beginFoo(int id); // clients in other processes can guess an ID
  void opFoo(int id);
  void recycleFoo(int id); // ownership not handled by type
}
```
Recommended: Individual interfaces

```
interface IManager {
   IFoo getFoo();
}

interface IFoo {
   void begin(); // clients in other processes can't guess a binder
   void op();
}
```
3. Don't mix one-way with two-way methods

[-Wmixed-oneway] Don't mix one-way with non-oneway methods, because it makes understanding the threading model complicated for clients and servers. Specifically, when reading client code of a particular interface, you need to look up for each method if that method will block or not.

4. Avoid returning status codes
Methods should avoid status codes as return values, since all AIDL methods have an implicit status return code. See ServiceSpecificException or EX_SERVICE_SPECIFIC. By convention, these values are defined as constants in an AIDL interface. More detailed information is in the Error handling section of AIDL backends.

5. Arrays as output parameters considered harmful
[-Wout-array] Methods having array output parameters, like void foo(out String[] ret) are usually bad because the output array size must be declared and allocated by the client in Java, and so the size of the array output cannot be chosen by the server. This undesirable behavior happens because of how arrays work in Java (they cannot be reallocated). Instead prefer APIs like String[] foo().

6. Avoid inout parameters
[-Winout-parameter] This can confuse clients because even in parameters look like out parameters.

7. Avoid out and inout @nullable non-array parameters
[-Wout-nullable] Since the Java backend doesn't handle @nullable annotation while other backends do, out/inout @nullable T may lead inconsistent behavior across backends. For example, non-Java backends can set an out @nullable parameter to null (in C++, setting it as std::nullopt) but the Java client can't read it as null.

Structured parcelables
1. When to use
Use structured parcelables where you have multiple data types to send.

Or, when you have a single data type but you expect that you will need to extend it in the future. For example, don't use String username. Use an extendable parcelable, like the following:

```
parcelable User {
   String username;
}
```

So that, in the future, you can extend it, as follows:

```
parcelable User {
   String username;
   int id;
}
```

2. Provide defaults explicitly
[-Wexplicit-default, -Wenum-explicit-default] Provide explicit defaults for fields.

Nonstructured parcelables
1. When to use
Nonstructured parcelables are available in Java with @JavaOnlyStableParcelable and in the NDK backend with @NdkOnlyStableParcelable. Usually, these are old and existing parcelables that can't be structured.

Constants and enums
1. Bitfields should use constant fields
Bitfields should use constant fields (for example, const int FOO = 3; in an interface).

2. Enums should be closed sets.
Enums should be closed sets. Note: only the interface owner can add enum elements. If vendors or OEMs need to extend these fields, an alternative mechanism is needed. Whenever possible, upstreaming vendor functionality should be preferred. However, in some cases, custom vendor values may be allowed through (though, vendors should have a mechanism in place to version this, perhaps AIDL itself, they shouldn't be able to conflict with each other, and these values shouldn't be exposed to 3rd party apps).

3. Avoid values like "NUM_ELEMENTS"
Since enums are versioned, values which indicate how many values are present should be avoided. In C++, this can be worked around with, enum_range<>. For Rust, use enum_values(). In Java, there's no solution yet.

Not recommended: Using numbered values

```
@Backing(type="int")
enum FruitType {
```

```
    APPLE = 0,
    BANANA = 1,
    MANGO = 2,
    NUM_TYPES, // BAD
}
```

4. Avoid redundant prefixes and suffixes

[-Wredundant-name] Avoid redundant or repetitive prefixes and suffixes in constants and enumerators.

Not recommended: Using a redundant prefix

```
enum MyStatus {
    STATUS_GOOD,
    STATUS_BAD // BAD
}
```

Recommended: Directly naming the enum

```
enum MyStatus {
    GOOD,
    BAD
}
```

FileDescriptor

[-Wfile-descriptor] The use of FileDescriptor as an argument or the return value of an AIDL interface method is highly discouraged. Especially, when the AIDL is implemented in Java, this might cause file descriptor leak unless carefully handled. Basically, if you accept a FileDescriptor, you need to close it manually when it is no longer used.

For native backends, you are safe because FileDescriptor maps to unique_fd which is auto-closeable. But regardless of the backend language you would use, it is wise to NOT use FileDescriptor at all because this will limit your freedom to change the backend language in the future.

Instead, use ParcelFileDescriptor, which is autocloseable.

Variable units

Make sure that variable units are included in the name so that their units are well-defined and understood without needed to reference documentation

Examples

```
long duration; // Bad
long durationNsec; // Good
long durationNanos; // Also good

double energy; // Bad
double energyMilliJoules; // Good

int frequency; // Bad
int frequencyHz; // Good
```

Timestamps must indicate their reference

Timestamps (in fact, all units!) must clearly indicate their units and reference points.

Examples

```
/**
 * Time since device boot in milliseconds
 */
long timestampMs;

/**
 * UTC time received from the NTP server in units of milliseconds
 * since January 1, 1970
 */
long utcTimeMs;
```

The AIDL style guide you've shared gives strong guidance on how to write **clean, extensible, and backward-compatible IPC interfaces**. Here's a breakdown and deeper explanation of the most important parts with examples:

✅ 1. Versioning

Every time a client or server is shipped (e.g., in a Mainline release), take a **snapshot** using:

```sh
m <interface-module-name>-freeze-api
```

Each snapshot = a version. This ensures backward compatibility and allows interface evolution.

🔁 Example

Initial version:

```aidl
parcelable User {
    String name;
}
```

New version:

```aidl
parcelable User {
    String name;
    int id;
}
```

Only safe if you freeze the old version. AIDL guarantees old clients still work.

✅ 2. Interface Design

⚠️ Avoid: Large manager interfaces with IDs

```aidl
interface IManager {
    int getSessionId();
    void op(int id);
}
```

- ❌ Error-prone
- ❌ Client may guess IDs and misuse

✅ Preferred: Subinterfaces

```aidl
interface IManager {
    ISession createSession();
}

interface ISession {
    void op();
}
```

- ✅ Safer (binders can't be guessed)
- ✅ Simpler object lifecycle

✅ 3. Method Parameters

❌ Avoid out/inout @nullable parameters

```aidl
void fetchData(out @nullable Data d); // BAD – not safe for Java
```

Java backend doesn't enforce nullable consistency. Use return values instead:

```aidl
@nullable Data fetchData(); // GOOD
```

✅ 4. Parcelables

⚠️ Use Structured Parcelables

Structured parcelables help versioning and maintainability.

✅ **Example:**

```aidl
parcelable User {
    String username;
}
```

❌ **Bad:**

```aidl
String username;
```

Because you can't extend it easily later.

Always provide explicit defaults:

```aidl
parcelable User {
    int id = 0;
    @nullable String name;
}
```

✅ 5. Enums and Constants

❌ Avoid:

```aidl
enum Status {
    STATUS_OK,
    STATUS_FAIL
}
```

✅ Better:

```aidl
enum Status {
    OK,
    FAIL
}
```

❌ Avoid enum count:

```aidl
NUM_ELEMENTS // Breaks versioning
```

✅ 6. Return status with exceptions, not codes

AIDL already has an implicit status return (via binder status code).

Use:

```aidl
@ServiceSpecificException
```

✅ 7. Arrays

❌ Don't do this:

```aidl
void getItems(out String[] items); // Client has to allocate first
```

✅ Do this:

```aidl
String[] getItems(); // Server allocates and returns
```

✅ 8. Naming Style

- Interface: **`IManager`**
- Parcelables: **`UserAccount`**
- Methods: **`getUserInfo()`**
- Enum constants: **`ENUM_VALUE`**
- Use units in variable names: **`timeMs`**, **`durationNanos`**, **`energyMilliJoules`**

✅ 9. FileDescriptors

Avoid using **`FileDescriptor`** directly.

❌ Bad:

```aidl
void openFd(in FileDescriptor fd); // Can leak
```

✅ Good:

```aidl
void openFd(in ParcelFileDescriptor fd); // Auto-closeable
```

✅ 10. Timestamp/Units

Always state **unit + reference point**.

```aidl
long uptimeMs;        // since boot
long utcTimeMs;       // since 1970 epoch
```

Would you like me to generate a **sample AIDL interface and parcelables** that follow this style guide?

24. Changes in aidl happen over the year till now

AIDL (Android Interface Definition Language) has evolved significantly over the years, especially to meet the increasing modularity, stability, and performance needs of Android (especially post-Treble). Here's a **year-by-year summary of major changes and improvements to AIDL**, including features added, deprecated practices, and backend evolutions:

📜 Timeline of Major AIDL Changes (Up to Android 14)

🔹 Android 7.0 (Nougat)

- `@nullable` annotation introduced
- Initial support for structured parcelables
- Support for `vec<T>`, arrays, enums

🔹 Android 8.0 (Oreo)

- **Project Treble** introduced:
  - **HIDL** launched as a separate mechanism for HALs
  - AIDL still used mainly for app-framework and system-service communication

---

🔹 Android 9.0 (Pie)

- Stabilization of AIDL framework usage
- Minor improvements in Java and C++ backends
- HIDL became primary for HALs

🔹 Android 10 (Q)

- **Hidden API support** with `@UnsupportedAppUsage`
- HIDL reaches maturity, but internal discussions began about **AIDL for HALs**

🔹 Android 11 (R) — Major Shift

- **Stable AIDL introduced**
  - Designed to replace HIDL for HALs
  - Supports system-vendor separation via **@VintfStability**
- **AIDL Backends Introduced:**
  - **Java**
  - **NDK**
  - **CPP**
  - **Rust** (early phase)
- **LazyServiceRegistrar** introduced:
  - Allows **dynamic (on-demand) AIDL services**
- `@Hide`, `@JavaOnlyStableParcelable`, and `@Backing` annotations added

🔹 Android 12 (S)

- **Fast Message Queue (FMQ) support for AIDL**
  - Via `AidlMessageQueue` for native IPC with low latency

- **Rust backend becomes more functional**
- `@RustDerive`, `@FixedSize` added
- **MQDescriptor<T, flavor>** added to support FMQ as first-class parcelables

---

🔹 Android 13 (T)

- `@nullable(heap=true)` added for recursive types
- `@JavaDefault` disables unused default impl generation to reduce size
- **Auto-generated equals/hashCode/toString** using `@JavaDerive`
- Improvements to **AIDL versioning and freezing system**
- `aidl_interface` module improvements to support metadata and multiple versions

---

🔹 Android 14 (U)

- **AIDL becomes preferred over HIDL** for all new HALs
- Enforcement of **structured types** for stable parcelables
- **FileDescriptor discouragement** becomes stronger
- Improvements to fuzzer tooling for AIDL (`fuzzService`)
- More `aidl_interface` Soong build system refinements (e.g., `backend`, `stability`, `versions`)
- Rust backend now considered stable for production use
- Extended annotations support for Rust and C++

🔮 What's Expected Beyond Android 14/15 (ongoing development)

- Deprecation of legacy Java-only or unstructured parcelables
- Tooling improvements for:
  - ABI verification across versions
  - Fuzzer automation per module
- More usage in updatable APEX modules (modular system components)
- **Language-independent interface IDL usage across all system processes**

📌 Summary of Key Feature Additions

Android Version	Key AIDL Changes
7.0	`@nullable`, basic annotations
8.0	Treble introduced; HIDL becomes HAL standard
11	Stable AIDL, `@VintfStability`, NDK/CPP/Rust backend, dynamic services
12	FMQ in AIDL, `AidlMessageQueue`, `@RustDerive`, `@FixedSize`
13	`@JavaDerive`, `@nullable(heap=true)`, better versioning tools
14	Enforced structured types, strong anti-pattern warnings, fuzzing support

---

Would you like a **visual timeline or table of this evolution**, or maybe a **comparison chart between HIDL vs AIDL over time**?

25. what is stabling in aidl

✅ What is "Stability" in AIDL?

In AIDL, **stability** refers to how safe and compatible a type or interface is to use **across system boundaries**, particularly **between system and vendor partitions** in Android.

Stability ensures:

- **Safe communication between different processes**, even across partitions (like system ↔ vendor).
- **Versioning and backward compatibility**, especially important for **modular system components** (like APEXes) and **updatable HALs**.
- That the **interface is frozen**, meaning once defined, it can't change in breaking ways.

🧩 Types of AIDL Stability

AIDL supports **four levels of stability**:

Stability	Description	Used For
vintf	Versioned and cross-partition (system ↔ vendor) stable	HALs, system-vendor communication
platform	Stable within the platform (system-side only)	System services
apex	Stable within APEX modules	Modular system components
local	No stability guaranteed (default)	Local-only services or prototypes

Each stability level controls **how and where** the interface or parcelable can be used.

🔒 `@VintfStability` Annotation

This annotation marks an **interface or data type** as **usable across system ↔ vendor partitions**. Without it, trying to use an interface across partitions results in a runtime or validation failure.

```aidl
@VintfStability
interface IMyHal {
    void doSomething();
}

@VintfStability
parcelable MyData {
    int value;
}
```

✅ Effect:

- Enforced by the binder framework.
- Required for HALs and other low-level APIs.
- Makes it possible to version interfaces across Android releases.

🧊 Interface Freezing and Versioning

Once a type is marked stable (like `vintf`), **you cannot change it arbitrarily**.

You must:

- Use the `aidl_interface` module in Soong
- Use `m <interface-name>-freeze-api` to snapshot the API
- Only allow **compatible changes** (e.g., add methods, not remove or change signatures)

🛠️ AOSP Example

```bp
aidl_interface {
    name: "android.hardware.example",
    srcs: ["IMyHal.aidl"],
    stability: "vintf",
    backend: {
        ndk: {
            enabled: true,
        },
        cpp: {
            enabled: true,
        },
    },
    versions: ["1"],
}
```

Later, to create version 2:

```sh
m android.hardware.example-freeze-api
```

This generates a snapshot of version 1, and allows you to start working on version 2 safely.

🚫 What Happens Without Stability?

If you don't define stability:

- The interface can't be used across system/vendor boundaries.
- You can't track ABI compatibility or enforce API changes.
- It's unsafe for updatable modules like APEX.

Summary

Term	Meaning
Stability	Guarantees how an AIDL interface or type behaves across updates/partitions
`@VintfStability`	Enables usage across system ↔ vendor
`aidl_interface`	A Soong module to manage versioning, stability, backends
Freeze API	Prevents accidental breaking changes in released interfaces

Would you like an example AIDL project with different stability types (e.g., **`vintf`**, **`apex`**, **`local`**) to see how they behave?