Beyond HIDL: Modernizing Android HALs with Rust & Stable AIDL

By Ankit S. & Jaimin K.
Silicon Signals Pvt. Ltd.

Speakers



Ankit Siddhapura
Technical Lead



Jaimin Kalariya
Senior Engineer

Agenda

01 Why HAL interfaces matter for Android's architecture

02 Android's HAL update requirements with version upgrade
HIDL vs AIDL — what changed

03 Will Rust become the future of Android HALs?
What is changing in HAL development

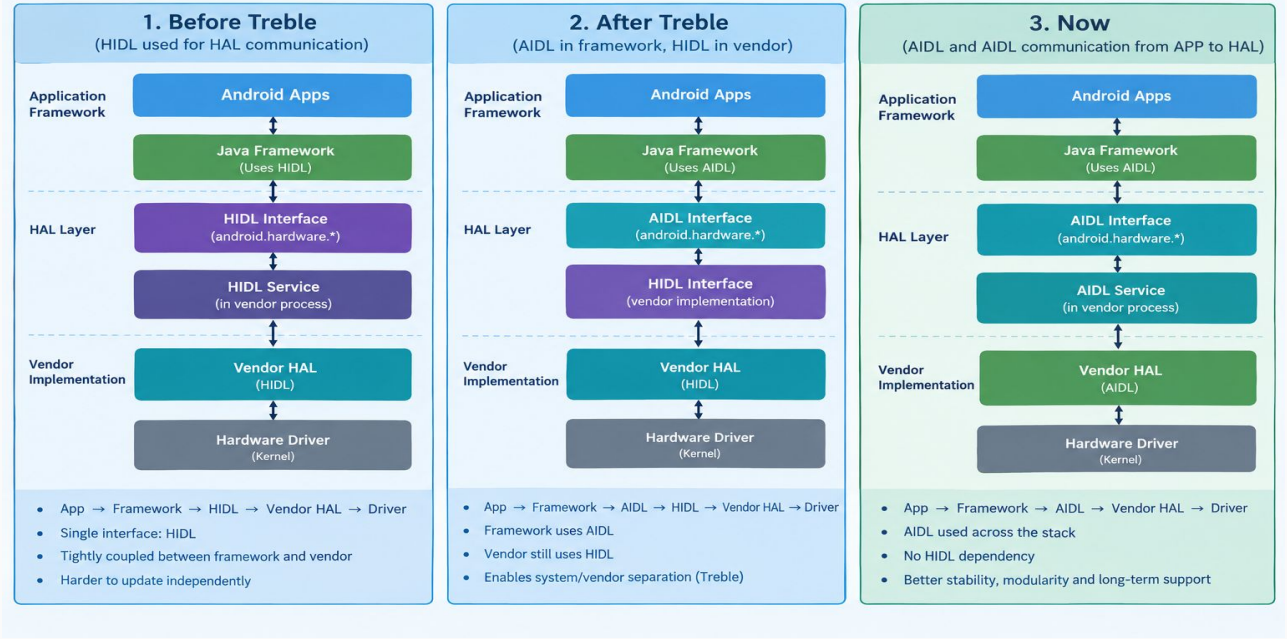
04 Q&A

Introduction

- Android upgrades bring new capabilities, but vendor HAL updates are required to fully support those features.
- AIDL is preferred for modern AOSP because it provides stable, versioned IPC with better framework–vendor compatibility and long-term maintainability.
- HIDL has no Rust backend at all, so a HAL frozen there isn't just missing new features; it's structurally unable to adopt Rust until someone does the AIDL port.
- With Stable AIDL providing the modern interface, Rust becomes the next step toward safer and more maintainable HAL implementations.

Where the HAL sits in the stack

→ The Hardware Abstraction Layer is the contract between the Android framework and vendor-supplied code.



Why Modernize Android HALs?

- Devices that launched on Android 8–12 shipped with HIDL Camera, Audio, Sensors, and other HALs — and many are still receiving OS upgrades today.
- OTA-able framework features from Android 13+ (new camera capabilities, extensions, streaming use cases) require an AIDL HAL underneath — no AIDL, no feature
- Newer framework compatibility matrices expect AIDL-declared services. A device stuck on HIDL risks failing VINTF checks on a future OS bump.
- Open source ROMs extend device life years past OEM support — but they can only track upstream AOSP if the vendor HAL keeps pace with AIDL.

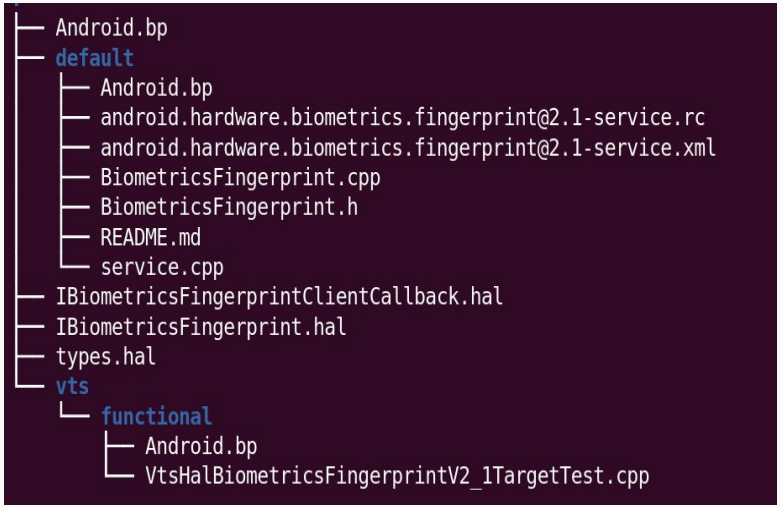
What changes in HAL - HIDL to AIDL

- Convert existing HIDL hal into AIDL using a tool - hidl2aidl
- Google provides a virtual Fingerprint AIDL interface as reference HAL, so device ports can implement the stable interface directly instead of converting device-specific HIDL.
- Update VINTF manifests, Both device manifest.xml (vendor) and framework compatibility matrix need the HAL entry moved from a hal format="hidl" block to hal format="aidl", with the new interface/instance name.
- Update SELinux policy & validate with VTS

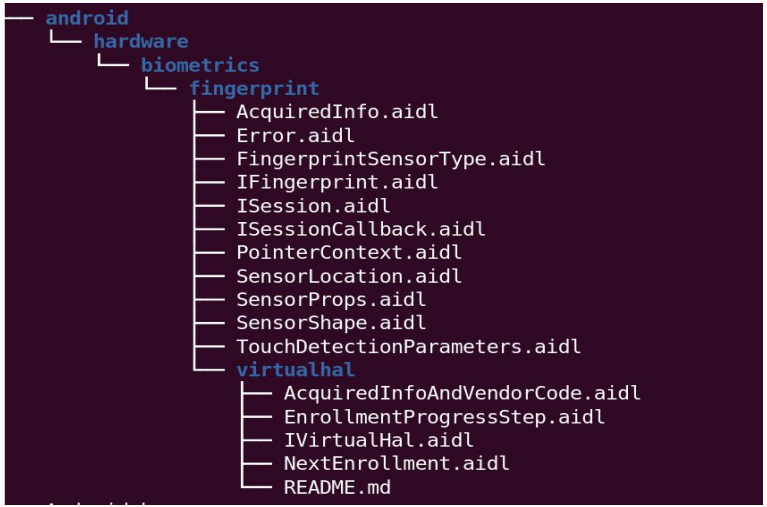
HAL Interfaces HIDL and AIDL

→ Example of fingerprint HAL interfaces.

HIDL



AIDL



Fingerprint HAL : HIDL to AIDL

→ Update existing interface for vendor implementation in hardware/interface/biometrics/fingerprint

Google's virtual HAL implementation

```
├── Android.bp
├── apex_file_contexts
├── apex_manifest.json
├── api
│   └── android.hardware.biometrics.fingerprint.VirtualProps-current.txt
├── FakeFingerprintEngine.cpp
├── FakeFingerprintEngineRear.cpp
├── FakeFingerprintEngineSide.cpp
├── FakeFingerprintEngineUdfps.cpp
├── FakeFingerprintEngineUdfpsUltraSonic.cpp
├── FakeLockoutTracker.cpp
├── FingerprintConfig.cpp
├── Fingerprint.cpp
├── fingerprint-default.rc
├── fingerprint-default.xml
├── fingerprint.sysprop
├── fingerprint-virtual.rc
├── include
│   ├── FakeFingerprintEngine.h
│   ├── FakeFingerprintEngineRear.h
│   ├── FakeFingerprintEngineSide.h
│   ├── FakeFingerprintEngineUdfps.h
│   ├── FakeFingerprintEngineUdfpsUltraSonic.h
│   ├── FakeLockoutTracker.h
│   ├── FingerprintConfig.h
│   ├── Fingerprint.h
│   ├── Session.h
│   └── VirtualHal.h
├── main.cpp
├── README.md
├── Session.cpp
├── tests
│   ├── FakeFingerprintEngineTest.cpp
│   ├── FakeFingerprintEngineUdfpsTest.cpp
│   ├── FakeLockoutTrackerTest.cpp
│   ├── SessionTest.cpp
│   └── VirtualHalTest.cpp
└── VirtualHal.cpp
```

Vendor Customization

```
├── Android.bp
├── FingerprintConfig.cpp
├── Fingerprint.cpp
├── fingerprint-default.rc
├── fingerprint-default.xml
├── FingerprintEngine.cpp
├── include
│   ├── Callable.h
│   ├── FakeFingerprintEngine.h
│   ├── FakeFingerprintEngineRear.h
│   ├── FakeFingerprintEngineSide.h
│   ├── FakeFingerprintEngineUdfps.h
│   ├── FakeLockoutTracker.h
│   ├── FingerprintConfig.h
│   ├── FingerprintEngine.h
│   ├── Fingerprint.h
│   ├── hardware
│   │   └── fingerprint.h
│   ├── Session.h
│   ├── VirtualHal.h
│   └── WorkerThread.h
├── main.cpp
├── README.md
├── Session.cpp
└── WorkerThread.cpp
```

Continue...

→ Device manifests can be extended using manifest fragments provided by HALs.

```
<manifest version="1.0" type="device">
    <hal format="aidl">
        <name>android.hardware.biometrics.fingerprint</name>
        <version>3</version>
        <fqname>IFingerprint/default</fqname>
    </hal>
</manifest>
```

→ Add AIDL support and remove HIDL entries for product packages and vintf device manifest.

→ SELinux updates

HIDL	AIDL
`hwbinder` class perms (`call` , `transfer` , `impersonate`)	`binder` class perms (`call` , `transfer` , `impersonate`)
`hwservice_manager` perms (`find` , `add`)	`service_manager` perms (`find` , `add`)
`/dev/hwbinder` device access	`/dev/binder` device access

Challenges for devices running in Android(in this HAL context)

- Not every HIDL HAL must be immediately migrated to AIDL. Newer Android versions can support a mixed HIDL/AIDL environment, depending on the HAL and Android release.

- **Vendor compatibility**: Older HIDL-based vendor partitions can cause VINTF mismatches, service failures, feature failures, or boot failures after a framework update.

- Vendor-proprietary libraries and components must also support the new AIDL interface. If proprietary libraries are still tightly coupled to HIDL interfaces, simply adding an AIDL HAL will not be sufficient.

- **64-bit transition**: Vendor components still dependent on 32-bit binaries may need updates to support the new AIDL-based implementation.

Rust Transition

Everything about HIDL-to-AIDL migration we've covered so far is also the foundation for the transition to Rust-based HALs coming up next..

AIDL opens the Rust path, but why Rust?

- Migrating from HIDL to stable AIDL enables the HAL implementation to be written in Rust.
- AIDL provides generated Rust bindings, making Rust practical for Android HAL development.
- Rust improves memory safety by preventing common native issues such as use-after-free, buffer overflows, and double frees.
- Rust modernizes the HAL implementation without changing Android's existing Binder, VINTF, and SELinux architecture.

Rust to C++ AIDL HAL — File Structure & Changes

hardware/interface/flashlight

```
.
└── aidl
    ├── aidl_api
    │   └── vendor.silicon.hardware.flashlight
    │       ├── 1
    │       │   └── vendor
    │       │       └── silicon
    │       │           └── hardware
    │       │               └── flashlight
    │       │                   └── IFlashlight.aidl
    │       └── current
    │           └── vendor
    │               └── silicon
    │                   └── hardware
    │                       └── flashlight
    │                           └── IFlashlight.aidl
    ├── Android.bp
    ├── default
    │   ├── Android.bp
    │   ├── src
    │   │   └── main.rs
    │   ├── vendor.silicon.hardware.flashlight-service.rc
    │   └── vendor.silicon.hardware.flashlight-service.xml
    └── vendor
        └── silicon
            └── hardware
                └── flashlight
                    └── IFlashlight.aidl
```

Rust HAL			C++ HAL	
Implementation file	`main.rs`	→	**Implementation file**	`main.cpp`
Build type	`rust_binary`	→	**Build type**	`cc_binary`
AIDL backend	`rust (enabled)`	→	**AIDL backend**	`cpp (enabled)`
Dependencies	`rustlibs` (libbinder_rs, etc.)	→	**Dependencies**	`c++ libs` (libbinder, libbase, liblog, etc.)

AIDL / VINTF / Java / RC / XML
UNCHANGED

AIDL contract and Rust build output

→ Stable AIDL generates the Rust HAL binding.

IFlashlight.aidl

```
package vendor.silicon.hardware.flashlight;

@VintfStability
interface IFlashlight {
    boolean setTorchMode(boolean enable);
    boolean getTorchMode();
}
```

Android.bp

```
aidl_interface {
    name: "vendor.silicon.hardware.flashlight",
    stability: "vintf",
    backend: {
        cpp: { enabled: false },
        rust: { enabled: true },
    },
    frozen: true,
}
```

Rust service and driver control

→ Rust service: Binder call to torch hardware

flashlight/aidl/default/src/main.rs

```rust
fn main() {
    logger::init(
        logger::Config::default()
            .with_tag_on_device("SiliconFlashlightHal")
            .with_max_level(log::LevelFilter::Info),
    );

    let service =
        BnFlashlight::new_binder(Flashlight::new(), BinderFeatures::default());

    binder::add_service(SERVICE_NAME, service.as_binder())
        .expect("Failed to register Silicon flashlight AIDL service");

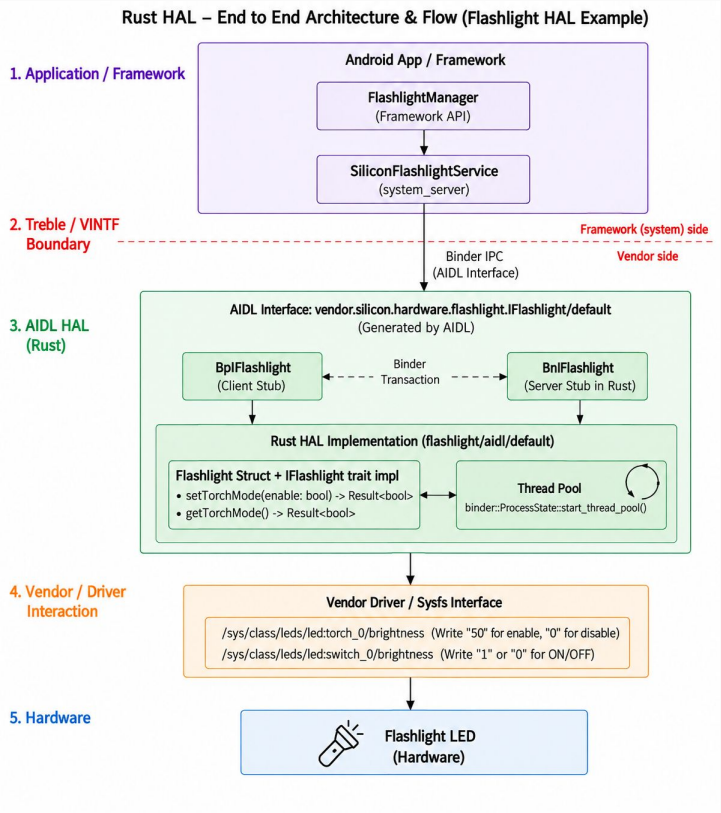
    log::info!("Silicon flashlight HAL registered");
    binder::ProcessState::start_thread_pool();
    binder::ProcessState::join_thread_pool();
}
```

```rust
const TORCH_BRIGHTNESS_PATH: &str =
    "/sys/class/leds/led:torch_0/brightness";
const TORCH_SWITCH_PATH: &str =
    "/sys/class/leds/led:switch_0/brightness";

const TORCH_BRIGHTNESS: &str = "50";

const SERVICE_NAME: &str =
    "vendor.silicon.hardware.flashlight.IFlashlight/default";
```

The Complete Flow



Common Challenges in Developing

- JNI — you don't need it just because the HAL is Rust. It's for Java/Kotlin calling native code directly, or bridging into existing C/C++ vendor libraries. Binder doesn't care what's on the other end.
- SELinux**:** Rust does not bypass SELinux; HALs still require proper domains**,** file_contexts**,** and service_contexts. Permissive mode is for bringup/testing, not production.
- Converting an old HAL — it’s not just changing the language. The interface stays the same, but you have to rebuild the implementation and rediscover the timing, ordering, and driver behavior handled by the old C++ code
- A panic kills the whole service, quietly. <.init> just respawns it in a loop — no dialog, no obvious signal. Grep for "panicked at" in logcat before chasing anything else.

Q&A

It’s a pleasure to be part of IndiaFOSS. Always open to learning, sharing, and contributing to the open-source community.

Any questions?
Ask away!

Give us a chance to address your doubts or concerns. Let’s open up the discussion with your ideas!

Scan QR for [flashlight Rust hal source]

Thank you!

Resources:

https://source.android.com/docs/core/architecture/aidl/aidl-hals

https://siliconsignals.io/blog/implementing-custom-hardware-abstraction-layer-hal-in-android-with-aidl-silicon-signals/

https://source.android.com/docs/setup/build/rust/building-rust-modules/android-rust-modules

https://android-review.googlesource.com/c/platform/hardware/interfaces/+/2628631



Got more questions?

Please reach out to

info@siliconsignals.io