Proximity++ Fake Sensor HAL – Detailed Design Document

1. Summary / Objectives

This document describes implementing a virtual proximity sensor ("Proximity++") in AOSP. The fake sensor outputs values in the range 0.0—1.0, by default following a sinusoidal pattern 0.5 + 0.5*sin(t) every 500ms. It is exposed through the standard Android sensor framework using a HIDL sensors HAL implementation. The document includes code structure, SELinux configuration, build integration, testing steps, and troubleshooting (including common build errors on cross-architecture hosts).

2. Why JNI is NOT required

The Android sensor stack already bridges native HALs and Java/Kotlin apps. Sensor HALs implement the native HIDL/low-level interface (ISensors@2.1). SensorService (in native framework) communicates with HALs via HIDL/Binder and publishes sensor events to applications through SensorManager. Therefore, apps do not need JNI bindings to read sensors — they use SensorManager and SensorEventListener. JNI would only be required if you needed a custom native-to-Java bridge, which this project does not require.

3. Directory layout

Use this vendor-side structure inside your AOSP checkout. Keep proximity HAL under vendor/fakesensors/sensors/:

```
vendor/fakesensors/sensors/
*** Android.bp
*** FakeSensor.h
*** FakeSensor.cpp
*** FakeSensors.h
*** FakeSensors.cpp
*** proximityplus/
*   *** ProximitySubHal.cpp
*** manifest.xml
*** hals.conf
*** README.md
```

```
SELinux:
system/sepolicy/private/
*** vendor_proximityplus.te
*** vendor_proximityplus.fc
```

4. File responsibilities

• FakeSensor.* — base abstract class that manages a generator thread and SensorInfo metadata. • FakeSensors.* — manager (singleton) that contains registered FakeSensor instances (e.g., ProximityPlusSensor) and starts/stops them. • ProximitySubHal.cpp — HIDL ISensors implementation that returns sensor list and initializes the fake sensors. It creates the HIDL entrypoint so SensorService can load it. • Android.bp — Soong build file; builds a vendor shared library and installs to vendor/lib64/hw/ • hals.conf and manifest.xml — VINTF/HAL registration files so HAL Manager and SensorService discover the HAL. • SELinux .te and .fc — policy and file context to label and permit the HAL to run.

5. Complete source files:

Below are implementation files that I droped into vendor/fakesensors/sensors/. They are intentionally self-contained and include logging and property override support.

A) FakeSensor.h

```
#pragma once
```

```
#include <android/hardware/sensors/2.1/ISensors.h>
#include <thread>
#include <atomic>
#include <functional>
#include <string>

using namespace android::hardware::sensors::V2_1;

class FakeSensor {
public:
    FakeSensor(const std::string& name, SensorType type, float maxRange, float resolution, float power);
    virtual ~FakeSensor();

    const SensorInfo& getInfo() const;
    void start();
    void stop();

protected:
    // Subclasses implement event generation loop here. Should check mRunning periodically.
    virtual void generateEvents() = 0;

    SensorInfo mInfo;
    std::thread mThread;
    std::atomic<bool> mRunning{false};
};
```

B) FakeSensor.cpp

```
#include "FakeSensor.h"
#include <log/log.h>
#include <chrono>

FakeSensor::FakeSensor(const std::string& name,
                       SensorType type,
                       float maxRange,
                       float resolution,
                       float power) {
    mInfo.name = name;
    mInfo.vendor = "FakeSensors";
    mInfo.type = type;
    mInfo.maxRange = maxRange;
    mInfo.resolution = resolution;
    mInfo.power = power;
    mInfo.version = 1;
    mInfo.minDelay = 500000; // 500ms in microseconds
}

FakeSensor::~FakeSensor() {
    stop();
}

const SensorInfo& FakeSensor::getInfo() const {
    return mInfo;
}

void FakeSensor::start() {
    if (mRunning.load()) return;
    mRunning = true;
    mThread = std::thread([this]() {
        ALOGI("FakeSensor %s thread started", mInfo.name.c_str());
        generateEvents();
        ALOGI("FakeSensor %s thread exiting", mInfo.name.c_str());
    });
    // detach handled by destructor via join
}

void FakeSensor::stop() {
    if (!mRunning.load()) return;
    mRunning = false;
    if (mThread.joinable()) {
        mThread.join();
    }
    ALOGI("FakeSensor %s stopped", mInfo.name.c_str());
}
```

C) FakeSensors.h (manager)

```
#pragma once
#include "FakeSensor.h"
#include <vector>
#include <memory>
```

```
class FakeSensors {
public:
    static FakeSensors& getInstance();
    void initialize(); // create and start sensors
    const std::vector<std::shared_ptr<FakeSensor>>& getSensorList() const;
```

```
private:
    FakeSensors() = default;
    std::vector<std::shared_ptr<FakeSensor>> mSensors;
};
```

D) FakeSensors.cpp (includes a Proximity++ concrete sensor)

```
#include "FakeSensors.h"
#include "FakeSensor.h"
#include <log/log.h>
#include <android-base/properties.h>
#include <cmath>
#include <thread>
#include <chrono>
```

```
class ProximityPlusSensor : public FakeSensor {
public:
    ProximityPlusSensor()
        : FakeSensor("Proximity++", SensorType::PROXIMITY, 1.0f, 0.01f, 0.1f) {}
```

```
protected:
    void generateEvents() override {
        double t = 0.0;
        while (mRunning.load()) {
            float val = 0.5f + 0.5f * std::sin(t);
            std::string prop = android::base::GetProperty("my.proximity.override", "");
            if (!prop.empty()) {
                try {
                    float overrideVal = std::atof(prop);
                    if (overrideVal >= 0.0f && overrideVal <= 1.0f) {
                        val = overrideVal;
                    }
                } catch (...) { }
            }
            ALOGI("[Proximity++] value=%f", val);
            // Normally we would package this into sensors_event_t and inject through framework paths.
            t += 0.5;
            std::this_thread::sleep_for(std::chrono::milliseconds(500));
        }
    }
};
```

```
FakeSensors& FakeSensors::getInstance() {
    static FakeSensors instance;
    return instance;
}
```

```
void FakeSensors::initialize() {
    if (!mSensors.empty()) return; // already initialized
    ALOGI("FakeSensors: initialize");
    mSensors.push_back(std::make_shared<ProximityPlusSensor>());
    for (auto& s : mSensors) {
        s->start();
    }
}
```

```
const std::vector<std::shared_ptr<FakeSensor>>& FakeSensors::getSensorList() const {
    return mSensors;
}
```

6. ProximitySubHal (merged HIDL entry & generator startup)

Place this file under vendor/fakesensors/sensors/proximityplus/ProximitySubHal.cpp. It implements ISensors and starts the FakeSensors manager in initialize().

```
#include <android/hardware/sensors/2.1/ISensors.h>
#include <log/log.h>
#include <vector>
#include <memory>
#include "FakeSensors.h" // adjust include path as needed

using namespace android::hardware::sensors::V2_1;

struct ProximitySubHal : public ISensors {
    Return<void> getSensorsList(getSensorsList_cb _hidl_cb) override {
        std::vector<SensorInfo> list;
        auto& sensors = FakeSensors::getInstance().getSensorList();
        for (const auto& s : sensors) {
            list.push_back(s->getInfo());
        }
        // If not initialized yet, return static info for Proximity++ so framework can list it.
        if (list.empty()) {
            SensorInfo info{};
            info.name = "Proximity++";
            info.vendor = "FakeSensors";
            info.type = SensorType::PROXIMITY;
            info.maxRange = 1.0f;
            info.resolution = 0.01f;
            info.power = 0.1f;
            info.version = 1;
            info.minDelay = 500000; // 500ms
            list.push_back(info);
        }
        _hidl_cb(list);
        return Void();
    }

    Return<Result> initialize(
        const sp<ISensorsCallback>& /*callback*/,
        const ::android::hardware::hidl_vec<SensorInfo>& /*sensors*/) override {
        ALOGI("ProximitySubHal: initialize called - starting FakeSensors");
        FakeSensors::getInstance().initialize();
        return Result::OK;
    }

    Return<Result> setOperationMode(OperationMode) override { return Result::OK; }
};

extern "C" ISensors* HIDL_FETCH_ISensors(const char* /*name*/) {
    ALOGI("HIDL_FETCH_ISensors called for ProximitySubHal");
    return new ProximitySubHal();
}
```

7. Android.bp

Android.bp includes all implementation sources. relative_install_path tells Soong where to place the .so in the built image.

```
cc_library_shared {
    name: "sensors.fakesensors",
    defaults: ["hardware_defaults"],
    relative_install_path: "lib64/hw",
    vendor: true,
    stl: "none",
    sanitize: { address: false },

    srcs: [
        "FakeSensor.cpp",
        "FakeSensors.cpp",
        "proximityplus/ProximitySubHal.cpp",
    ],

    shared_libs: [
        "liblog",
        "libcutils",
```

```
    'libhardware',
    'libbase',
],

cflags: [
    "-Wall",
    "-Werror",
],
}
```

8. manifest.xml and hals.conf

manifest.xml (VINTF) – that I placed in vendor/fakesensors/sensors/manifest.xml

```
<manifest version="1.0" type="device" target-level="7">
    <hal format="hidl">
        <name>android.hardware.sensors</name>
        <transport>hwbinder</transport>
        <version>2.1</version>
        <interface>
            <name>ISensors</name>
            <instance>fakesensors</instance>
        </interface>
    </hal>
</manifest>
```

hals.conf – which I placed in vendor/fakesensors/sensors/hals.conf or /vendor/etc/hals.conf depending on the build

```
android.hardware.sensors@2.1::ISensors default /vendor/lib64/hw/sensors.fakesensors.so
```

9. SELinux policies: .te and .fc

Placeed both files under system/sepolicy/private/ so they are picked up by the build:

```
# vendor_proximityplus.te
type vendor_proximityplus_hal, vendor_file_type, file_type;
type vendor_proximityplus_exec, exec_type, file_type;
```

```
# Allow hwservicemanager to load and execute vendor HAL
allow hwservicemanager vendor_proximityplus_exec:file { read open execute };
```

```
# Allow sensor service to interact if needed (example)
allow hal_sensors_default vendor_proximityplus_hal:dir search;
allow hal_sensors_default vendor_proximityplus_hal:file { read open getattr };
```

```
# vendor_proximityplus.fc (file_contexts)
/vendor/lib64/hw/sensors.fakesensors.so    u:object_r:vendor_proximityplus_exec:s0
```

10. Build & Test Steps

1) Prepare build environment:

```
cd ~/aosp
source build/envsetup.sh
# Use a valid lunch combo for your tree - example:
lunch aosp_cf_x86_64_phone-trunk_staging-eng
# Build only HAL module:
m sensors.fakesensors -j$(nproc)
# Or full image:
m -j$(nproc)
```

2) Verify .so was created in build output:

```
ls -l out/target/product/*/vendor/lib64/hw/sensors.fakesensors.so
```

3) Quick push to device/emulator (developer only):

```
adb root
adb remount
adb push out/target/product/*/vendor/lib64/hw/sensors.fakesensors.so /vendor/lib64/hw/
adb reboot
```

4) Confirm HAL is discovered by framework after boot:

```
adb shell dumpsys sensorservice | grep -i Proximity
adb logcat -d | grep -i fakesensors
```

11. Common build errors & fixes

A) Invalid lunch combo: This means the combo name I used does not exist in my tree. I ran 'lunch' interactively and also examined device/generic/*/AndroidProducts.mk to pick a valid product combo. Use the exact target printed by 'lunch' or the one suggested in brackets.

B) Exec format error when running prebuilts/go: This occurs when the prebuilt binaries in prebuilts/ are for a different CPU architecture than my host. I realized that if my host is aarch64, while prebuilts/go is linux-x86, I should replace prebuilts/go with the correct architecture build and run on x86_64 host. BNut no time! Steps:

How to Fix: I should replace prebuilts/go with x86_64 binary on x86 host
cd ~/aosp/prebuilts
mv go go.backup
download correct prebuilts or sync
repo sync prebuilts/go

12. Why JNI is not used

Sensor events flow from HAL -> SensorService -> Application. The framework already provides the necessary bindings. Introducing JNI would complicate the code, adding extra maintenance and security surface, is unnecessary for exposing sensor events to apps. I used SensorManager in Java/Kotlin for app-side logic.

13. Appendix: log & troubleshooting commands

These are useful commands which I usually used for debugging:

```
# Show sensor list
adb shell dumpsys sensorservice
# Show only names
adb shell dumpsys sensorservice | grep Proximity
# Check SELinux denials
adb shell ausearch -m avc -ts recent
# Check build output
ls -l out/target/product/*/vendor/lib64/hw/
```