OpenAI

Configuration_Property

1. Configuration overview

bookmark_border
AOSP offers the following options for storing configuration information on a device:

System properties
Hardware abstraction layer (HAL) properties
System config XML files
Resource overlays (static and runtime)
System properties
System properties are string key/value pairs stored in the build.prop global dictionary. System properties are system-wide resources that are easy to use and have a low performance overhead. When using system properties, you don't need to use interprocess communication (IPC) even if a system property is shared across multiple processes. However, system properties are similar to global variables and can be harmful when misused. The misuse of system properties can result in issues such as security vulnerabilities and apps becoming inaccessible to users. Before using system properties to store configuration information, consider the other configuration options.

For further information on system properties, see Add system properties

Note: Previous to Android 10, AOSP used a ConfigStore HAL to store system properties. ConfigStore HAL is deprecated and should no longer be used. For information on the ConfigStore HAL, refer to ConfigStore HAL.
HAL properties
When the source of truth for a configuration is from a hardware component on a device, the HAL for the hardware must provide the information for that component. Define a new HAL method in the existing HAL for accessing the configuration. For further information on developing a HAL, see AIDL for HALs.

Note: Don't configure the HAL to use system properties as a side-channel communication mechanism for HALs.
System config XML files
When the configuration data is static but complicated (structured), consider using XML or other such formats for the configuration data. Ensure that the file schema remains stable. For XML files, you can use xsd_config to keep the schema stable, and to take advantage of an autogenerated XML parser.

Resource overlay
You can use resource overlays to customize a product. There are two types of resource overlays:

Standard resource overlay used to customize a product at build time. Foris information on standard resource overlays, see Customizing the build with resource overlays.

Runtime resource overlay (RRO) is used to change the resource values of a target package at runtime. For example, an app installed on the system image might change its behavior based upon the value of a resource. Rather than hardcoding the resource value at build time, an RRO installed on a different partition can change the values of the app's resources at runtime. For more information on RROs, see Change the value of an app's resources at runtime.

Here's a **simple breakdown** of the different **configuration storage options in AOSP**:

🔧 1. System Properties

- **What it is**: Key-value string pairs (like `ro.debuggable=1`) stored in `build.prop`.
- **Use case**: Global, simple flags or values used across Android (e.g., checking if the build is debuggable).
- **Pros**:
  - Fast and accessible from all processes.
  - No IPC overhead.
- **Cons**:
  - Can be misused like global variables.
  - May cause **security issues** or **unexpected behavior** if overused.
  - Deprecated for HALs since Android 10 (don't use system properties inside HALs).

👉 Use **only if** others (like HALs or XMLs) don't fit.

🧱 2. HAL Properties

- **What it is**: When configuration comes directly from **hardware**, it should be exposed via HAL.
- **Use case**: If your hardware component (like a camera or sensor) has settings (e.g., max resolution), expose it through its **HAL interface**.
- **Pros**:
  - Clean, hardware-specific, encapsulated access.
- **Cons**:
  - Should not use system properties as a **backdoor** (side channel).

👉 **Preferred** when config is hardware-originated.

📁 3. System Config XML Files

- **What it is**: Static, structured configuration files in **XML format**.
- **Use case**: Good for complex configs like **device capabilities**, **permissions**, **component settings**.
- **Pros**:
  - Easy to manage structure.
  - You can use **XSD validation** (`xsd_config`) and autogenerated parsers.
- **Cons**:
  - Not dynamic — must be defined at build time.

👉 Use for structured, **static configuration data**.

🎨 4. Resource Overlays

- **What it is**: A way to override app/system resources (e.g., colors, dimensions, strings) without modifying the base code.
- Two types:
  - ✅ **Static Resource Overlay** (at **build time**)
  - 🕒 **Runtime Resource Overlay (RRO)** (at **runtime**, even from another partition)
- **Use case**: Modify appearance or behavior of system components or apps without recompiling them.
- **Pros**:
  - Safe and modular.
  - RRO allows **customization after build**, useful for vendors.
- **Cons**:
  - Limited to resource values, not logic.

👉 Use to **customize system or app behavior/resources** cleanly.

✅ Summary Table

Option	When to Use	Dynamic?	Structured?	Safe for Vendors?
System Properties	Simple global flags, used rarely and wisely	No	No	⚠️ Use with caution
HAL Properties	Config originates from hardware	Yes	Depends	✅ Yes
System Config XML	Static, complex structured configs (capabilities)	No	Yes	✅ Yes
Static Resource Overlay	Build-time customizations	No	Yes	✅ Yes
RRO (Runtime Overlay)	Runtime customization (e.g., change app behavior)	Yes	Yes (resources)	✅ Yes

---

Let me know if you'd like examples for any of them (e.g., `build.prop`, HAL AIDL method, config XML file, or overlay RRO structure).

2. Implement system properties as APIs

bookmark_border
System properties provide a convenient way to share information, usually configurations, system-wide. Each partition can use its own system properties internally. A problem can happen when properties are accessed across partitions, such as /vendor accessing /system-defined properties. Since Android 8.0, some partitions, such as /system, can be upgraded, while /vendor is left unchanged. Because system properties are just a global dictionary of string key-value pairs with no schema, it's difficult to stabilize properties. The /system partition could change or remove properties that the /vendor partition depends on without any notice.

Starting with the Android 10 release, system properties accessed across partitions are schematized into Sysprop Description files, and APIs to access properties are generated as concrete functions for C++ and Rust, and classes for Java. These APIs are more convenient to use because no magic strings (such as ro.build.date) are needed for access, and because they can be statically typed. ABI stability is also checked at build time, and the build breaks if incompatible changes happen. This check acts as explicitly defined interfaces between partitions. These APIs can also provide consistency between Rust, Java and C++.

Define system properties as APIs
Define system properties as APIs with Sysprop Description files (.sysprop), which use a TextFormat of protobuf, with the following schema:

```
// File: sysprop.proto

syntax = "proto3";

package sysprop;

enum Access {
 Readonly = 0;
 Writeonce = 1;
 ReadWrite = 2;
}

enum Owner {
 Platform = 0;
 Vendor = 1;
 Odm = 2;
}

enum Scope {
 Public = 0;
 Internal = 2;
}

enum Type {
 Boolean = 0;
 Integer = 1;
 Long = 2;
 Double = 3;
 String = 4;
 Enum = 5;
 UInt = 6;
 ULong = 7;

 BooleanList = 20;
 IntegerList = 21;
 LongList = 22;
 DoubleList = 23;
 StringList = 24;
 EnumList = 25;
 UIntList = 26;
 ULongList = 27;
}

message Property {
```

```
  string api_name = 1;
  Type type = 2;
  Access access = 3;
  Scope scope = 4;
  string prop_name = 5;
  string enum_values = 6;
  bool integer_as_bool = 7;
  string legacy_prop_name = 8;
}

message Properties {
  Owner owner = 1;
  string module = 2;
  repeated Property prop = 3;
}
```

One Sysprop Description file contains one properties message, which describes a set of properties. The meaning of its fields are as follows.

Field	Meaning
owner	Set to the partition that owns the properties: Platform, Vendor, or Odm.
module	Used to create a namespace (C++) or static final class (Java) in which generated APIs are placed. For example, com.android.sysprop.BuildProperties will be namespace com::android::sysprop::BuildProperties in C++, and the BuildProperties class in the package in com.android.sysprop in Java.
prop	List of properties.

The meanings of the Property message fields are as follows.

Field	Meaning
api_name	The name of the generated API.
type	The type of this property.
access	Readonly: Generates getter API only Writeonce, ReadWrite: Generates getter and setter APIs Note: Properties with the prefix ro. may not use ReadWrite access.
scope	Internal: Only the owner can access. Public: Everyone can access, except for NDK modules.
prop_name	The name of the underlying system property, for example ro.build.date.
enum_values	(Enum, EnumList only) A bar(|)-separated string that consists of possible enum values. For example, value1|value2.
integer_as_bool	(Boolean, BooleanList only) Make setters use 0 and 1 instead of false and true.
legacy_prop_name	(optional, Readonly properties only) The legacy name of the underlying system property. When calling getter, the getter API tries to read prop_name and uses legacy_prop_name if prop_name doesn't exist. Use legacy_prop_name when deprecating an existing property and moving to a new property.

Each type of property maps to the following types in C++, Java, and Rust.

Type	C++	Java	Rust
Boolean	std::optional<bool>	Optional<Boolean>	bool
Integer	std::optional<std::int32_t>	Optional<Integer>	i32
UInt	std::optional<std::uint32_t>	Optional<Integer>	u32
Long	std::optional<std::int64_t>	Optional<Long>	i64
ULong	std::optional<std::uint64_t>	Optional<Long>	u64
Double	std::optional<double>	Optional<Double>	f64
String	std::optional<std::string>	Optional<String>	String
Enum	std::optional<{api_name}_values>	Optional<{api_name}_values>	{ApiName}Values
T List	std::vector<std::optional<T>>	List<T>	Vec<T>

Here's an example of a Sysprop Description file defining three properties:

```
# File: android/sysprop/PlatformProperties.sysprop

owner: Platform
module: "android.sysprop.PlatformProperties"
prop {
    api_name: "build_date"
    type: String
    prop_name: "ro.build.date"
    scope: Public
    access: Readonly
}
prop {
```

```
    api_name: "date_utc"
    type: Integer
    prop_name: "ro.build.date_utc"
    scope: Internal
    access: Readonly
}
prop {
    api_name: "device_status"
    type: Enum
    enum_values: "on|off|unknown"
    prop_name: "device.status"
    scope: Public
    access: ReadWrite
}
```

Define system properties libraries

You can now define sysprop_library modules with Sysprop Description files. sysprop_library serves as an API for C++, Java and Rust. The build system internally generates one rust_library, one java_library and one cc_library for each instance of sysprop_library.

```
// File: Android.bp
sysprop_library {
    name: "PlatformProperties",
    srcs: ["android/sysprop/PlatformProperties.sysprop"],
    property_owner: "Platform",
    vendor_available: true,
}
```

You must include API lists files in the source for API checks. To do this, create API files and an api directory. Put the api directory in the same directory as Android.bp. The API filenames are <module_name>-current.txt, <module_name>-latest.txt. <module_name>-current.txt holds the API signatures of current source codes, and <module_name>-latest.txt holds the latest frozen API signatures. The build system checks whether the APIs are changed by comparing these API files with generated API files at build time and emits an error message and instructions to update current.txt file if current.txt doesn't match with the source codes. Here's an example directory and file organization:

```
├── api
│   ├── PlatformProperties-current.txt
│   └── PlatformProperties-latest.txt
└── Android.bp
```

Note: The lower version of sysprop_library might have a different set of API signature files, which are current.txt, removed.txt, system-current.txt, system-removed.txt, test-current.txt, and test-removed.txt. If that's the case, you can update them by using the m update-api command.

Rust, Java and C++ client modules can link against sysprop_library to use generated APIs. The build system creates links from clients to generated C++, Java and Rust libraries, thus giving clients access to generated APIs.

```
java_library {
    name: "JavaClient",
    srcs: ["foo/bar.java"],
    libs: ["PlatformProperties"],
}

cc_binary {
    name: "cc_client",
    srcs: ["baz.cpp"],
    shared_libs: ["libPlatformProperties"],
}

rust_binary {
    name: "rust_client",
    srcs: ["src/main.rs"],
    rustlibs: ["libplatformproperties_rust"],
}
```

Note that the Rust library name is generated by converting the sysprop_library name to lowercase, replacing . and - with _, and then prepending lib and appending _rust.

In the preceding example, you could access defined properties as follows.

Rust example:

```
use platformproperties::DeviceStatusValues;

fn foo() -> Result<(), Error> {
  // Read "ro.build.date_utc". default value is -1.
  let date_utc = platformproperties::date_utc()?.unwrap_or_else(-1);

  // set "device.status" to "unknown" if "ro.build.date" is not set.
  if platformproperties::build_date()?.is_none() {
    platformproperties::set_device_status(DeviceStatusValues::UNKNOWN);
  }

  …
}
Java example:


import android.sysprop.PlatformProperties;

…

static void foo() {
    …
    // read "ro.build.date_utc". default value is -1
    Integer dateUtc = PlatformProperties.date_utc().orElse(-1);

    // set "device.status" to "unknown" if "ro.build.date" is not set
    if (!PlatformProperties.build_date().isPresent()) {
        PlatformProperties.device_status(
            PlatformProperties.device_status_values.UNKNOWN
        );
    }
    …
}
…
C++ example:


#include <android/sysprop/PlatformProperties.sysprop.h>
using namespace android::sysprop;

…

void bar() {
    …
    // read "ro.build.date". default value is "(unknown)"
    std::string build_date = PlatformProperties::build_date().value_or("(unknown)");

    // set "device.status" to "on" if it's "unknown" or not set
    using PlatformProperties::device_status_values;
    auto status = PlatformProperties::device_status();
    if (!status.has_value() || status.value() == device_status_values::UNKNOWN) {
        PlatformProperties::device_status(device_status_values::ON);
    }
    …
}
…
```

Here's a **simple explanation and step-by-step guide** to **implement and use Android system properties as typed APIs** using `sysprop_library`.

🧩 Why This Exists

Starting **Android 10**, system properties are made **type-safe** and **partition-safe** using:

- `.sysprop` **files** (protobuf format) to define properties.
- **Generated APIs** in **Java**, **C++**, and **Rust** to access them.
- **ABI checks** to prevent incompatible changes across partitions like `/vendor` and `/system`.

🧱 Step-by-Step: Define System Properties as APIs

✅ Step 1: Create a `.sysprop` File

Example: `android/sysprop/PlatformProperties.sysprop`

```protobuf
owner: Platform
module: "android.sysprop.PlatformProperties"
prop {
    api_name: "build_date"
    type: String
    prop_name: "ro.build.date"
    scope: Public
    access: Readonly
}
prop {
    api_name: "device_status"
    type: Enum
    prop_name: "device.status"
    enum_values: "on|off|unknown"
    scope: Public
    access: ReadWrite
}
```

✅ Step 2: Define the `sysprop_library` in `Android.bp`

```bp
sysprop_library {
    name: "PlatformProperties",
    srcs: ["android/sysprop/PlatformProperties.sysprop"],
    property_owner: "Platform",
    vendor_available: true,
}
```

✅ Step 3: Create API Description Files

Create these in an `api/` directory beside `Android.bp`:

```markdown
.
├── Android.bp
└── api
    ├── PlatformProperties-current.txt
    └── PlatformProperties-latest.txt
```

Use:

```bash
m update-api
```

to generate or update these automatically.

✅ Step 4: Use the Generated APIs

▶️ Java Example

```java
import android.sysprop.PlatformProperties;

void foo() {
    String buildDate = PlatformProperties.build_date().orElse("(unknown)");

    if (!PlatformProperties.build_date().isPresent()) {
        PlatformProperties.device_status(
            PlatformProperties.device_status_values.UNKNOWN);
    }
}
```

▶️ C++ Example

```cpp
#include <android/sysprop/PlatformProperties.sysprop.h>
using namespace android::sysprop;

void bar() {
    std::string buildDate = PlatformProperties::build_date().value_or("(unknown)");

    auto status = PlatformProperties::device_status();
    if (!status.has_value() || status.value() == PlatformProperties::device_status_values::UNKNOWN) {
        PlatformProperties::device_status(PlatformProperties::device_status_values::ON);
    }
}
```

▶️ Rust Example

```rust
use platformproperties::DeviceStatusValues;

fn foo() -> Result<(), std::io::Error> {
    let date_utc = platformproperties::date_utc()?.unwrap_or(-1);

    if platformproperties::build_date()?.is_none() {
        platformproperties::set_device_status(DeviceStatusValues::UNKNOWN);
    }

    Ok(())
}
```

🔗 Step 5: Link Clients to the API

Java Client

```bp
java_library {
    name: "JavaClient",
    srcs: ["MyClient.java"],
    libs: ["PlatformProperties"],
}
```

C++ Client

```bp
cc_binary {
    name: "cc_client",
    srcs: ["my_client.cpp"],
    shared_libs: ["libPlatformProperties"],
}
```

Rust Client

```bp
rust_binary {
    name: "rust_client",
    srcs: ["main.rs"],
```

```
    rustlibs: ["libplatformproperties_rust"],
}
```

🔐 ABI Stability and Safety

- ABI is checked **at build time** using the `-current.txt` vs. generated signatures.
- Prevents breakages when `/system` updates independently of `/vendor`.

🔁 Summary Table

Concept	Description
`.sysprop` file	Defines schema, type, access, visibility
`sysprop_library`	Soong module to generate C++, Java, Rust APIs
`current.txt` / `latest.txt`	Tracks API changes for ABI enforcement
Access from client code	Use clean, typed APIs — no raw string keys
Tools	`m update-api` to refresh API files

Let me know if you want a **real working example directory structure**, or help writing your **own custom sysprop** module and client.

3. Add system properties

bookmark_border
This page provides a canonical method for adding or defining system properties in Android, with guidelines for refactoring existing system properties. Ensure that you use the guidelines when you refactor, unless you have a strong compatibility issue that dictates otherwise.

Step 1: Define the system property
When you add a system property, decide on a name for the property, and associate it with an SELinux property context. If there's no appropriate existing context, create a new one. The name is used when accessing the property; the property context is used to control accessibility in terms of SELinux. Names can be any string, but AOSP recommends that you follow a structured format to make them clear.

Property name
Use this format with snake_case casing:

[{prefix}.]{group}[.{subgroup}]*.{name}[.{type}]
Use either "" (omitted), ro (for properties set only once), or persist (for properties that persist across reboots) for the element prefix.

Caveats
Use ro only when you're certain that you don't need prefix to be writable in the future. ** Don't specify the ro prefix.** Instead, rely on sepolicy to make prefix read-only (in other words, writable only by init).

Use persist only when you're certain that the value must be persisted across reboots, and that using the system properties is your only option.

Google strictly reviews the system properties that have either ro or persist properties.

The term group is used to aggregate related properties. It's intended to be a subsystem name similar in use to audio or telephony. Don't use ambiguous or overloaded terms such as sys, system, dev, default, or config.

It's common practice to use the name of the domain type of a process that has exclusive read or write access to the system properties. For example, for the system properties to which the vold process has write access, it's common to use vold (the name of the domain type for the process) as the group name.

If needed, add subgroup to further categorize properties, but avoid ambiguous or overloaded terms to describe this element. (You can also have more than one subgroup.)

Many group names have already been defined. Check the system/sepolicy/private/property_contexts file and use existing group names where possible, instead of making new ones. The following table provides examples of frequently used group names.

Domain	Group (and subgroup)
bluetooth related	bluetooth
sysprops from kernel cmdline	boot
sysprops that identify a build	build
telephony related	telephony
audio related	audio
graphics related	graphics
vold related	vold

The following defines the use of name and type in the previous regex example.

[{prefix}.]{group}[.{subgroup}]*.{name}[.{type}]

name identifies a system property within a group.

type is an optional element that clarifies the type or intent of the system property. For example, instead of naming a sysprop as audio.awesome_feature_enabled or just audio.awesome_feature, rename it as audio.awesome_feature.enabled to reflect the system property type and intent.

There's no specific rule about what the type must be; these are usage recommendations:

enabled: Use if the type is a boolean system property that's used to turn a feature on or off.
config: Use if the intent is to clarify that the system property doesn't represent a dynamic state of the system; it represents a preconfigured value (for example, a read-only thing).
List: Use if it's a system property whose value is a list.

Timeoutmillis: Use if it's a system property for a timeout value in units of ms.
Examples:

persist.radio.multisim.config
drm.service.enabled
Property context
The new SELinux property context scheme allows for finer granularity and more descriptive names. Similar to what's used for property names, AOSP recommends the following format:

{group}[_{subgroup}]*_prop
The terms are defined as follows:

group and subgroup have the same meaning as defined for the previous sample regex. For example, vold_config_prop signifies properties which are configurations from a vendor and meant to be set by vendor_init, while vold_status_prop or just vold_prop signifies properties which are to expose the current status of vold.

When naming a property context, choose names that reflect the general usage of the properties. In particular, avoid the following types of terms:

Terms which look too general and ambiguous, such as sys, system, default.
Terms that directly encode accessibility: such as exported, apponly, ro, public, private.
Prefer name usages like vold_config_prop to exported_vold_prop, or vold_vendor_writable_prop.

Type
A property type can be one of the following as listed in the table.

Type	Definition
Boolean	true or 1 for true, false or 0 for false
Integer	signed 64-bit integer
Unsigned integer	unsigned 64-bit integer
Double	double-precision floating point
String	any valid UTF-8 string
enum	values can be any valid UTF-8 string without whitespaces
List of above	A comma (,) is used as the delimiter

The integer list [1, 2, 3] is stored as 1,2,3
Internally, all properties are stored as strings. You can enforce the type by specifying it as a property_contexts file. For more information, see property_contexts in Step 3.

Step 2: Determine required accessibility levels
There are four helper macros that define a property.

Accessibility type	Meaning
system_internal_prop	Properties which are used only in /system
system_restricted_prop	Properties which are read outside /system, but not written
system_vendor_config_prop	Properties which are read outside /system, and written only by vendor_init
system_public_prop	Properties which are read and written outside /system

Scope the access to system properties as narrowly as possible. In the past, broad access has resulted in app breakage and security vulnerabilities. Consider the following questions when scoping:

Does this system property need to be persisted? (if so, why?)
Which process should have read access to this property?
Which process should have write access to this property?
Use the preceding questions and the following decision tree as tools for determining an appropriate scope for access.

Decision tree for determining the scope of access

Figure 1. Decision tree for determining scope of access to system properties

Step 3: Add to system/sepolicy
When accessing sysprop, SELinux controls the accessibility of processes. After you determine what level of accessibility is required, define property contexts under system/sepolicy, along with additional allow and neverallow rules about what the processes are (and aren't) allowed to read or write.

First, define the property context in the system/sepolicy/public/property.te file. If the property is system-internal, define it in the system/sepolicy/private/property.te file. Use one of thesystem_[accessibility]_prop([context]) macros that provides the accessibility required of your system property. This is an example for the system/sepolicy/public/property.te file:

system_public_prop(audio_foo_prop)
system_vendor_config_prop(audio_bar_prop)
Example to add in the system/sepolicy/private/property.te file:

system_internal_prop(audio_baz_prop)
Second, grant read and (or) write access to the property context. Use set_prop and get_prop macros to grant access, in either the system/sepolicy/public/{domain}.te or system/sepolicy/private/{domain}.te file. Use private whenever possible; public is suitable only if the set_prop or get_prop macro affects any domains outside the core domain.

Example, in the system/sepolicy/private/audio.te file:

set_prop(audio, audio_foo_prop)
set_prop(audio, audio_bar_prop)
Example, in the system/sepolicy/public/domain.te file:

get_prop(domain, audio_bar_prop)
Third, add some neverallow rules to further reduce the accessibility that's scoped by the macro. For example, assume that you've used system_restricted_prop because your system properties must be read by vendor processes. If the read access isn't required by all vendor processes, and it's only required by a certain set of processes (such as vendor_init), prohibit the vendor processes that don't need the read access.

Use the following syntax for restricting write and read access:

To restrict write access:

neverallow [domain] [context]:property_service set;
To restrict read access:

neverallow [domain] [context]:file no_rw_file_perms;
Place neverallow rules in the system/sepolicy/private/{domain}.te file if the neverallow rule is bound to a specific domain. For broader neverallow rules, use general domains such as these wherever appropriate:

system/sepolicy/private/property.te
system/sepolicy/private/coredomain.te
system/sepolicy/private/domain.te
In the system/sepolicy/private/audio.te file, place the following:

```
neverallow {
    domain -init -audio
} {audio_foo_prop audio_bar_prop}:property_service set;
```

In the system/sepolicy/private/property.te file, place the following:

```
neverallow {
    domain -coredomain -vendor_init
} audio_prop:file no_rw_file_perms;
```

Note that {domain -coredomain} captures all vendor processes. So {domain -coredomain -vendor_init} means "all vendor processes except vendor_init."

Finally, associate a system property with the property context. This ensures that the access that's granted and the neverallow rules that are applied to property contexts are applied to actual properties. To do this, add an entry to the property_contexts file, a file that describes mapping between system properties and property contexts. In this file, you can specify either a single property, or a prefix for properties to be mapped into a context.

This is the syntax for mapping a single property:

[property_name] u:object_r:[context_name]:s0 exact [type]
This is the syntax for mapping a prefix:

[property_name_prefix] u:object_r:[context_name]:s0 prefix [type]
You can optionally specify the type of the property, which can be one of the following:

bool
int
uint
double
enum [list of possible values...]
string (Use string for list properties.)
Ensure that every entry has its designated type whenever possible, as type is enforced when setting property. The following example shows how to write a mapping:

binds a boolean property "ro.audio.status.enabled"
to the context "audio_foo_prop"
ro.audio.status.enabled u:object_r:audio_foo_prop:s0 exact bool

binds a boolean property "vold.decrypt.status"
to the context "vold_foo_prop"
The property can only be set to one of these: on, off, unknown
vold.decrypt.status u:object_r:vold_foo_prop:s0 exact enum on off unknown

binds any properties starting with "ro.audio.status."
to the context "audio_bar_prop", such as
"ro.audio.status.foo", or "ro.audio.status.bar.baz", and so on.
ro.audio.status. u:object_r:audio_bar_prop:s0 prefix
When an exact entry and a prefix entry conflict, the exact entry takes precedence. For more examples, see system/sepolicy/private/property_contexts.

Step 4: Determine stability requirements
Stability is another aspect of system properties, and it differs from accessibility. Stability is about whether or not a system property can be changed (for example renamed, or even removed) in the future. This is particularly important as Android OS becomes modular. With Treble, the system, vendor, and product partitions can be updated independently of one another. With Mainline, some parts of the OS are modularized as updatable modules (in APEXes or APKs).

If a system property is for use across updatable pieces of software, for example across system and vendor partitions, it must be stable. However, if it's used only within, for example, a specific Mainline module, you can change its name, type, or property contexts, and even remove it.

Ask the following questions to determine the stability of a system property:

Is this system property intended to be configured by partners (or configured differently per device)? If yes, it must be stable.
Is this AOSP-defined system property intended to be written to or read from code (not process) that exists in non-system partitions like vendor.img or product.img? If yes, it must be stable.
Is this system property accessed across Mainline modules or across a Mainline module and the non-updatable part of the platform? If yes, it must be stable.
For the stable system properties, formally define each as an API and use the API to access the system property, as explained in Step 6.

Note: AOSP recommends that you define a system property as an API even if it doesn't need to be stable. Doing so clearly enumerates all the system properties that your module owns in a single view, and simplifies the process of promoting an unstable system property to a stable one. (The mechanism for distinguishing between stable and unstable system property definitions is explained in Step 6.)
Step 5: Set properties at build time
Set properties at build time with makefile variables. Technically, the values are built into {partition}/build.prop. Then init reads {partition}/build.prop to set the properties. There are two sets of such variables: PRODUCT_{PARTITION}_PROPERTIES and TARGET_{PARTITION}_PROP.

PRODUCT_{PARTITION}_PROPERTIES contains a list of property values. The syntax is {prop}={value} or {prop}?={value}.

{prop}={value} is a normal assignment which ensures that {prop} is set to {value}; only one such assignment is possible per a single property.

{prop}?={value} is an optional assignment; {prop} sets to {value} only if there aren't any {prop}={value} assignments. If multiple optional assignments exist, the first one wins.

sets persist.traced.enable to 1 with system/build.prop

PRODUCT_SYSTEM_PROPERTIES += persist.traced.enable=1

sets ro.zygote to zygote32 with system/build.prop
but only when there are no other assignments to ro.zygote
optional are useful when giving a default value to a property
PRODUCT_SYSTEM_PROPERTIES += ro.zygote?=zygote32

sets ro.config.low_ram to true with vendor/build.prop
PRODUCT_VENDOR_PROPERTIES += ro.config.low_ram=true
TARGET_{PARTITION}_PROP contains a list of files, which is directly emitted to {partition}/build.prop. Each file contains a list of {prop}={value} pairs.

```
# example.prop

ro.cp_system_other_odex=0
ro.adb.secure=0
ro.control_privapp_permissions=disable

# emits example.prop to system/build.prop
TARGET_SYSTEM_PROP += example.prop
```

For more detail, see build/make/core/sysprop.mk.

Note: build.prop from vendor partitions and odm partitions are written under the context for vendor_init, and build.prop from other partitions are written under the context for init.
Step 6: Access properties at runtime
Properties can be read and written at runtime.

Init scripts
Init script files (usually *.rc files) can read a property by ${prop} or ${prop:-default}, can set an action which runs whenever a property becomes a specific value, and can write the properties using the setprop command.

```
# when persist.device_config.global_settings.sys_traced becomes 1,
# set persist.traced.enable to 1
on property:persist.device_config.global_settings.sys_traced=1
   setprop persist.traced.enable 1

# when security.perf_harden becomes 0,
# write /proc/sys/kernel/sample_rate to the value of
# debug.sample_rate. If it's empty, write -100000 instead
on property:security.perf_harden=0
   write /proc/sys/kernel/sample_rate ${debug.sample_rate:-100000}
```

Note: similar to build.prop, rc files under the vendor and odm partitions read and write using the same context as vendor_init.
getprop and setprop shell commands
You can use the getprop or setprop shell commands, respectively, to read or write the properties. For more details, invoke getprop --help or setprop --help.

```
$ adb shell getprop ro.vndk.version
$
$ adb shell setprop security.perf_harden 0
```

Sysprop as API for C++/Java/Rust
With sysprop as API, you can define system properties and use auto-generated API which are concrete and typed. Setting scope with Public also makes generated APIs available to modules across boundaries, and ensures API stability. Here's a sample of a .sysprop file, an Android.bp module, and C++, Java and Rust code using them.

```
# AudioProps.sysprop
# module becomes static class (Java) / namespace (C++) for serving API
module: "android.sysprop.AudioProps"
# owner can be Platform or Vendor or Odm
owner: Platform
# one prop defines one property
prop {
   prop_name: "ro.audio.volume.level"
```

```
    type: Integer
    scope: Public
    access: ReadWrite
    api_name: "volume_level"
}
...

// Android.bp
sysprop_library {
    name: "AudioProps",
    srcs: ["android/sysprop/AudioProps.sysprop"],
    property_owner: "Platform",
}

// Rust, Java and C++ modules can link against the sysprop_library
rust_binary {
    rustlibs: ["libaudioprops_rust"],
    ...
}

java_library {
    static_libs: ["AudioProps"],
    ...
}

cc_binary {
    static_libs: ["libAudioProps"],
    ...
}

// Rust code accessing generated API.
// Get volume. Use 50 as the default value.
let vol = audioprops::volume_level()?.unwrap_or_else(50);

// Java codes accessing generated API
// get volume. use 50 as the default value.
int vol = android.sysprop.AudioProps.volume_level().orElse(50);
// add 10 to the volume level.
android.sysprop.AudioProps.volume_level(vol + 10);

// C++ codes accessing generated API
// get volume. use 50 as the default value.
int vol = android::sysprop::AudioProps::volume_level().value_or(50);
// add 10 to the volume level.
android::sysprop::AudioProps::volume_level(vol + 10);
```

For more information, see Implement system properties as APIs.

C/C++, Java, and Rust low-level property functions and methods
When possible, use Sysprop as API even though low-level C/C++ or Rust functions or low-level Java methods are available to you.

libc, libbase, and libcutils offer C++ system property functions. libc has the underlying API, while the libbase and libcutils functions are wrappers. If it's possible, use the libbase sysprop functions; they're the most convenient, and host binaries can use the libbase functions. For more details, see sys/system_properties.h (libc), android-base/properties.h (libbase), and cutils/properties.h (libcutils).

The android.os.SystemProperties class offers Java system property methods.

The rustutils::system_properties module offers Rust system property functions and types.

Appendix: Add vendor-specific properties
Partners (including Googlers working in the context of Pixel development) want to define hardware-specific (or device-specific) system properties. Vendor-specific properties are partner-owned properties that are unique to their own hardware or device, not to the platform. As these are hardware or device dependent, they're meant to be used within the /vendor or /odm partitions.

Since Project Treble, the platform properties and vendor properties have been completely split to keep them from conflicting. The following describes how to define vendor properties, and tells which vendor properties must always be used.

Namespace on property and context names
All vendor properties must start with one of the following prefixes to prevent collision between them and the properties of other partitions.

ctl.odm.
ctl.vendor.
ctl.start$odm.
ctl.start$vendor.
ctl.stop$odm.
ctl.stop$vendor.
init.svc.odm.
init.svc.vendor.
ro.odm.
ro.vendor.
odm.
persist.odm.
persist.vendor.
vendor.
Note that ro.hardware. is allowed as a prefix, but for only compatibility. Don't use it for normal properties.

The following examples all use one of the preceding listed prefixes:

vendor.display.primary_red
persist.vendor.faceauth.use_disk_cache
ro.odm.hardware.platform
All vendor property contexts must start with vendor_. This is also for compatibility. The following are examples:

vendor_radio_prop.
vendor_faceauth_prop.
vendor_usb_prop.
It's the vendor's responsibility to name and maintain properties, so follow the format suggested in Step 2, in addition to the vendor namespaces requirements.

Vendor-specific SEPolicy rules and property_contexts
Vendor properties can be defined by vendor_internal_prop macro. Put the vendor-specific rules you define in the BOARD_VENDOR_SEPOLICY_DIRS directory. For example, suppose that you're defining a vendor faceauth property in coral.

In the BoardConfig.mk file (or in any BoardConfig.mk includes), put the following:

BOARD_VENDOR_SEPOLICY_DIRS := device/google/coral-sepolicy
In the device/google/coral-sepolicy/private/property.te file, put the following:

vendor_internal_prop(vendor_faceauth_prop)
In the device/google/coral-sepolicy/private/property_contexts file, put the following:

vendor.faceauth.trace u:object_r:vendor_faceauth_prop:s0 exact bool
Limitations of vendor properties
Because the system and product partitions can't depend on the vendor, never allow the vendor properties to be accessed from the system, system-ext, or product partitions.

Appendix: Rename existing properties
When you must deprecate a property and move to a new one, use Sysprop as APIs to rename your existing properties. This maintains backward compatibility by specifying both the legacy name and the new property name. Specifically, you can set the legacy name by the legacy_prop_name field in the .sysprop file. The generated API tries to read prop_name, and uses legacy_prop_name if prop_name doesn't exist.

For example, the following steps rename awesome_feature_foo_enabled to foo.awesome_feature.enabled.

In the foo.sysprop file

module: "android.sysprop.foo"
owner: Platform
prop {
  api_name: "is_awesome_feature_enabled"

type: Boolean
scope: Public
access: Readonly
prop_name: "foo.awesome_feature.enabled"
legacy_prop_name: "awesome_feature_foo_enabled"
}
In C++ code

// is_awesome_feature_enabled() reads "foo.awesome_feature.enabled".
// If it doesn't exist, reads "awesome_feature_foo_enabled" instead
using android::sysprop::foo;

bool enabled = foo::is_awesome_feature_enabled().value_or(false);
Note the following caveats:

First, you can't change the type of the sysprop. For example, you can't make an int prop into a string prop. You can only change the name.

Second, only the read API falls back to the legacy name. The write API doesn't fall back. If the sysprop is a writable one, you can't rename it.

Here's a **step-by-step summary and practical guide** to **adding new Android system properties** properly — including naming, access control (SELinux), stability, and usage via sysprop APIs.

🧱 Step-by-Step Guide to Add a System Property

✅ Step 1: Name the Property Correctly

Use structured format:

```text
[{prefix}.]{group}[.{subgroup}]*.{name}[.{type}]
```

🔹 Prefix options:

Prefix	Meaning
none	Writable by init
`ro.`	Read-only (set once by init)
`persist.`	Persistent across reboots

⚠️ **Avoid hardcoding** `ro.` — use `init` and SELinux to enforce read-only behavior instead.

🔹 Good Examples:

- `ro.audio.volume.level` (readonly int)
- `persist.vendor.camera.config.mode` (persistent string)
- `vendor.gps.mode.enabled` (vendor-specific bool)

✅ Step 2: Define the SELinux Property Context

This controls **which process can read/write** the property.

🔹 Use helper macros:

```c
// system/sepolicy/public/property.te or private/property.te
system_public_prop(audio_config_prop)
system_internal_prop(audio_debug_prop)
vendor_internal_prop(vendor_faceauth_prop)
```

🔹 Give access:

```c
// system/sepolicy/private/audio.te
set_prop(audio, audio_config_prop)
get_prop(audio, audio_config_prop)
```

🔹 **Add neverallow rules:**

```c
neverallow { domain -init -audio } audio_config_prop:property_service set;
neverallow { domain -coredomain -vendor_init } audio_config_prop:file no_rw_file_perms;
```

✅ Step 3: Map Property to Context

Edit **`property_contexts`** file.

🔹 **Example mappings:**

```text
ro.audio.volume.level     u:object_r:audio_config_prop:s0 exact int
persist.vendor.gps.mode   u:object_r:vendor_gps_prop:s0 exact string
ro.audio.volume.          u:object_r:audio_config_prop:s0 prefix int
```

- 🔸 Use **`exact`** for full name, **`prefix`** for group.

✅ Step 4: Set Value at Build Time

Use **`PRODUCT_*_PROPERTIES`** or **`TARGET_*_PROP`**.

🔹 **In `device.mk`:**

```make
PRODUCT_SYSTEM_PROPERTIES += ro.audio.volume.level=50
PRODUCT_VENDOR_PROPERTIES += persist.vendor.gps.mode=high
```

🔹 **Or use property files:**

```make
# props.prop
ro.my.prop=1
ro.gps.timeout=1000

# Android.mk
TARGET_SYSTEM_PROP += props.prop
```

✅ Step 5: Determine Stability

Ask:

- Is it accessed across partitions (system ↔ vendor)?
- Is it used by partners or OEM config?
- Is it accessed from APEX or another module?

If yes → **must be stable**, and should be **defined as Sysprop API**.

✅ Step 6: (Optional but Recommended) Define Property via Sysprop API

Create a **`.sysprop`** file:

🔹 **Example: `android/sysprop/AudioProps.sysprop`**

```protobuf
owner: Platform
module: "android.sysprop.AudioProps"

prop {
  api_name: "volume_level"
  type: Integer
  prop_name: "ro.audio.volume.level"
  scope: Public
  access: ReadWrite
}
```

🔹 **Add to Android.bp**

```bp
sysprop_library {
    name: "AudioProps",
    srcs: ["android/sysprop/AudioProps.sysprop"],
    property_owner: "Platform",
    vendor_available: true,
}
```

🔹 **Create `api/AudioProps-current.txt` and `-latest.txt` (run `m update-api`)**

✅ Step 7: Access at Runtime

🔹 **Via init `.rc`:**

```rc
on property:persist.vendor.feature.enabled=1
    setprop vendor.service.started 1
```

🔹 **From shell:**

```bash
adb shell getprop ro.audio.volume.level
adb shell setprop persist.vendor.gps.mode high
```

🔹 **Via Java:**

```java
int volume = AudioProps.volume_level().orElse(50);
AudioProps.volume_level(volume + 10);
```

🔹 **Via C++:**

```cpp
int vol = android::sysprop::AudioProps::volume_level().value_or(50);
```

🔹 **Via Rust:**

```rust
let vol = audioprops::volume_level()?.unwrap_or_else(|| 50);
```

🧪 Appendices

✅ Rename an Existing Property (maintain backward compatibility)

Use **`legacy_prop_name`** in **`.sysprop`**:

```
protobuf

prop {
  api_name: "is_feature_enabled"
  type: Boolean
  prop_name: "foo.feature.enabled"
  legacy_prop_name: "feature_enabled_legacy"
}
```

✅ Add Vendor Properties

- Prefix: **`vendor.`**, **`persist.vendor.`**, **`ro.vendor.`**, etc.
- Context name: **`vendor_<group>_prop`**
- Property SELinux rules and context in **`BOARD_VENDOR_SEPOLICY_DIRS`**

🧠 Summary Checklist

Task	Done?
Choose meaningful structured name	✅
Define SELinux property context	✅
Map property to context	✅
Add access and neverallow rules	✅
Add build-time value	✅
Define as Sysprop API (if needed)	✅
Use via API or shell/init script	✅

Let me know if you want:

- A **concrete example commit** for AOSP.
- **SEPolicy rule templates** for your use-case.
- Help defining your **custom vendor property**.

4. ↪
✅ Step 2: **Define the SELinux Property Context** This controls **which process can read/write** the property. #### 🔹 Use helper macros: c CopyEdit `// system/sepolicy/public/property.te or private/property.te system_public_prop(audio_config_prop) system_internal_prop(audio_debug_prop) vendor_internal_prop(vendor_faceauth_prop)` #### 🔹 Give access: c CopyEdit `// system/sepolicy/private/audio.te set_prop(audio, audio_config_prop) get_prop(audio, audio_config_prop)` #### 🔹 Add neverallow rules: c CopyEdit `neverallow { domain -init -audio } audio_config_prop:property_service set; neverallow { domain -coredomain -vendor_init } audio_config_prop:file no_rw_file_perms;`
explain in simple words with example

Let's break down **Step 2: Define the SELinux Property Context** into **very simple words with a full example**.

🎯 What Is This Step About?

This step is about **telling Android's security system (SELinux)**:

- "Which system property belongs to whom?"
- "Which process is allowed to **read or write** this property?"
- "Who should **never** be allowed to touch it?"

🧠 Think of It Like This:

Imagine the system property is a **locker**.

- You need to **label** the locker (`property context`).
- Then decide who has the **key to open (read)** or **write into it** (`set_prop`, `get_prop`).
- And finally say **who must never even touch it** (`neverallow`).

✅ Example in Simple Words

🔧 Let's say you add a new property:

```text
ro.audio.volume.level
```

This is a **readonly system property** that tells what the default audio volume is.

👣 Step-by-Step Example

🔹 1. Define Property Context

> 📁 File: `system/sepolicy/public/property.te`

```c
system_public_prop(audio_volume_prop)
```

This means:

> "I am creating a property context called `audio_volume_prop`, and it can be read by apps or services outside `system` (public)."

🔹 2. Give Read/Write Access

> 📁 File: `system/sepolicy/private/audio.te`

```c
get_prop(audio, audio_volume_prop)
```

This means:

> "Allow the **`audio`** process to read this property."

💡 Note:

- If your property is writable (not **`ro.`**), you would also use:

  ```c
  set_prop(audio, audio_volume_prop)
  ```

🔹 3. Add Neverallow Rules

> 📁 File: **`system/sepolicy/private/audio.te`**

```c
neverallow { domain -init -audio } audio_volume_prop:property_service set;
```

This means:

> "No one is allowed to write this property except the **`audio`** process or **`init`**."

> 📁 File: **`system/sepolicy/private/property.te`**

```c
neverallow { domain -coredomain -vendor_init } audio_volume_prop:file no_rw_file_perms;
```

This means:

> "No vendor process (except vendor_init) can read or write this property file."

🎁 BONUS: Add Mapping

> 📁 File: **`system/sepolicy/private/property_contexts`**

```text
ro.audio.volume.level u:object_r:audio_volume_prop:s0 exact int
```

This line connects:

- Your actual property name (**`ro.audio.volume.level`**)
- To the SELinux context (**`audio_volume_prop`**) you just defined.

🧪 Final Outcome

You now have:

- A secure property: **`ro.audio.volume.level`**
- Clearly labeled with **`audio_volume_prop`**
- Only the **`audio`** service can read it.
- No other process can accidentally write it or misuse it.

Would you like a real AOSP commit example for this too?

5. ↪
✅ Step 4: **Set Value at Build Time** Use `PRODUCT_*_PROPERTIES` or `TARGET_*_PROP`. #### 🔹 In `device.mk`: make CopyEdit `PRODUCT_SYSTEM_PROPERTIES += ro.audio.volume.level=50 PRODUCT_VENDOR_PROPERTIES += persist.vendor.gps.mode=high` #### 🔹 Or use property files: make CopyEdit `# props.prop ro.my.prop=1 ro.gps.timeout=1000 # Android.mk TARGET_SYSTEM_PROP += props.prop`
explain this in simple with example

Sure! Let's explain **Step 4: Set Value at Build Time** in **very simple words with examples**.

🎯 What's the Goal?

When you're building Android, you may want to set some system properties **in advance**, so that when the device boots, the system already knows:

- Default audio volume
- Whether GPS is enabled
- Device mode, etc.

This step is about **how to set those values at build time** (not at runtime).

🧠 Think of it like:

You're baking a cake (your Android image). You want to mix in some **settings** (properties) into the recipe **before baking**.

🧩 Two Ways to Do It:

✅ 1. Use `PRODUCT_*_PROPERTIES` in `device.mk`

This is **the simplest way**. Just list your properties there.

🔧 Example

> 📄 File: `device/myvendor/mydevice/device.mk`

```make
# Set audio volume level in system image
PRODUCT_SYSTEM_PROPERTIES += ro.audio.volume.level=50

# Set GPS mode in vendor image
PRODUCT_VENDOR_PROPERTIES += persist.vendor.gps.mode=high
```

💡 This will:

- Set `ro.audio.volume.level` inside `system/build.prop`
- Set `persist.vendor.gps.mode` inside `vendor/build.prop`

✅ 2. Use a `.prop` file and include it with `TARGET_*_PROP`

This is better when you have **many properties** to keep clean and organized.

🔧 Step-by-step Example

📄 Step 1: Create a `.prop` file

> 📄 File: `props.prop`

```text
ro.my.prop=1
ro.gps.timeout=1000
```

📄 Step 2: Include this file in your Android.mk

> 📄 File: **`Android.mk`**

```make
TARGET_SYSTEM_PROP += props.prop
```

This tells the build system:

> “Hey, take all the properties from **`props.prop`** and put them in **`system/build.prop`**.”

💡 When to Use Which?

Method	When to Use
`PRODUCT_*_PROPERTIES`	For 1–5 simple properties
`.prop` file + **`TARGET_*_PROP`**	For 5+ or grouped properties

📦 Where Does It Go?

Property	Ends up in
`PRODUCT_SYSTEM_PROPERTIES`	**`system/build.prop`**
`PRODUCT_VENDOR_PROPERTIES`	**`vendor/build.prop`**
`TARGET_SYSTEM_PROP += xyz.prop`	**`system/build.prop`**

✅ Final Device Output

After the build, if you run:

```bash
adb shell getprop
```

You will see:

```markdown
[ro.audio.volume.level]: [50]
[persist.vendor.gps.mode]: [high]
[ro.gps.timeout]: [1000]
```

Would you like me to generate a real example **`.prop`** file and corresponding **`Android.mk`** + **`device.mk`** structure?

6. Implement Config File Schema API

bookmark_border
The Android platform contains many XML files for storing config data (for example, audio config). Many of the XML files are in the vendor partition, but they're read in the system partition. In this case, the schema of the XML file serves as the interface across the two partitions, and therefore the schema must be explicitly specified and must evolve in a backward-compatible manner.

Before Android 10, the platform didn't provide mechanisms to require specifying and using the XML schema, or to prevent incompatible changes in the schema. Android 10 provides this mechanism, called Config File Schema API. This mechanism consists of a tool called xsdc and a build rule called xsd_config.

The xsdc tool is an XML Schema Document (XSD) compiler. It parses an XSD file describing the schema of an XML file and generates Java and C++ code. The generated code parses XML files that conform to the XSD schema into a tree of objects, each of which models an XML tag. XML attributes are modeled as fields of the objects.

The xsd_config build rule integrates the xsdc tool into the build system. For a given XSD input file, the build rule generates Java and C++ libraries. You can link the libraries to the modules where the XML files that conform to the XSD are read and used. You can use the build rule for your own XML files used across the system and vendor partitions.

Build Config File Schema API
This section describes how to build Config File Schema API.

Configure the xsd_config build rule in Android.bp
The xsd_config build rule generates the parser code with the xsdc tool. The xsd_config build rule's package_name property determines the package name of the generated Java code.

Example xsd_config build rule in Android.bp:

```
xsd_config {
  name: "hal_manifest",
  srcs: ["hal_manifest.xsd"],
  package_name: "hal.manifest",
}
```

Example directory structure:

```
├── Android.bp
├── api
│   ├── current.txt
│   ├── last_current.txt
│   ├── last_removed.txt
│   └── removed.txt
└── hal_manifest.xsd
```

The build system generates an API list using the generated Java code and checks the API against it. This API check is added to DroidCore and executed at m -j.

Create API lists files
The API checks require API lists files in the source code.

The API lists files include:

current.txt and removed.txt check whether the APIs are changed by comparing with generated API files at build time.
last_current.txt and last_removed.txt check whether the APIs are backward compatible by comparing with API files.
To create the API lists files:

Create empty lists files.
Run the command make update-api.
Use generated parser code
To use the generated Java code, add : as a prefix to the xsd_config module name in the Java srcs property. The package of the generated Java code is the same as the package_name property.

```
java_library {
    name: "vintf_test_java",
    srcs: [
        "srcs/**/*.java"
        ":hal_manifest"
    ],
}
```

To use the generated C++ code, add the xsd_config module name to the generated_sources and generated_headers properties. And add libxml2 to static_libs or shared_libs, since libxml2 is required in generated parser code. The namespace of the generated C++ code is the same as the package_name property. For example, if the xsd_config module name is hal.manifest, the namespace is hal::manifest.

```
cc_library{
    name: "vintf_test_cpp",
    srcs: ["main.cpp"],
    generated_sources: ["hal_manifest"],
    generated_headers: ["hal_manifest"],
    shared_libs: ["libxml2"],
}
```

Use the parser

To use the Java parser code, use the XmlParser#read or read{class-name} method to return the class of the root element. Parsing happens at this time.

```
import hal.manifest.*;

...

class HalInfo {
    public String name;
    public String format;
    public String optional;
    ...
}

void readHalManifestFromXml(File file) {
    ...
    try (InputStream str = new BufferedInputStream(new FileInputStream(file))) {
        Manifest manifest = XmlParser.read(str);
        for (Hal hal : manifest.getHal()) {
            HalInfo halinfo;
            HalInfo.name = hal.getName();
            HalInfo.format = hal.getFormat();
            HalInfo.optional = hal.getOptional();
            ...
        }
    }
    ...
}
```

To use the C++ parser code, first include the header file. The name of the header file is the package name with periods (.) converted to underscores (_). Then use the read or read{class-name} method to return the class of the root element. Parsing happens at this time. The return value is a std::optional<>.

```
include "hal_manifest.h"

...
using namespace hal::manifest

struct HalInfo {
    public std::string name;
    public std::string format;
    public std::string optional;
    ...
};
```

```
void readHalManifestFromXml(std::string file_name) {
    ...
    Manifest manifest = *read(file_name.c_str());
    for (Hal hal : manifest.getHal()) {
        struct HalInfo halinfo;
        HalInfo.name = hal.getName();
        HalInfo.format = hal.getFormat();
        HalInfo.optional = hal.getOptional();
        ...
    }
    ...
}
All the APIs provided to use the parser are in api/current.txt. For uniformity, all element and attribute names are converted to camel case (for example, ElementName) and used as the corresponding variable, method, and class name. The class of the parsed root element can be obtained using the read{class-name} function. If there is only one root element, then the function name is read. The value of a parsed subelement or attribute can be obtained using the get{variable-name} function.

Generate parser code
In most cases, you don't need to run xsdc directly. Use the xsd_config build rule instead, as described in Configuring the xsd_config build rule in Android.bp. This section explains the xsdc command line interface, just for completeness. This might be useful for debugging.

You must give the xsdc tool the path to the XSD file, and a package. The package is a package name in Java code and a namespace in C++ code. The options to determine whether the generated code is Java or C are -j or -c, respectively. The -o option is the path of the output directory.


usage: xsdc path/to/xsd_file.xsd [-c] [-j] [-o <arg>] [-p]
 -c,--cpp           Generate C++ code.
 -j,--java          Generate Java code.
 -o,--outDir <arg>  Out Directory
 -p,--package       Package name of the generated java file. file name of
                    generated C++ file and header
Example command:


$ xsdc audio_policy_configuration.xsd -p audio.policy -j
Was this helpful?
```

Absolutely! Let's break down **Config File Schema API** in **simple words** with a **real-world example** so you understand it clearly.

✅ What Problem Is It Solving?

Android uses many **XML configuration files** (e.g., audio settings, HAL manifests, etc.) spread across system and vendor partitions.

🤔 The problem before Android 10:

- These XML files had **no enforced schema**.
- One side (e.g., system) could break things if the file format changed, because the other side (e.g., vendor) wouldn't know.

📦 Solution in Android 10:

Use **XSD (XML Schema Definition)** to define the structure of the XML file — like a contract.
Then use a tool called `xsdc` to:

- Generate **Java/C++ classes** from that schema.
- Make it easy and **safe to read XML files** in code.
- Automatically **check for API/schema compatibility**.

🧪 Real-Life Analogy:

Think of an **XML config** like a Google Form response.

Now imagine two teams:

- **Team A (vendor)** fills out the form (writes the XML).
- **Team B (system)** reads it.

Without rules, Team A might change the questions any time → Team B's program breaks.

With XSD + `xsdc`, the structure is **strictly defined**, and Team B can generate **reliable code** to read the XML.

🔧 Step-by-Step Example

Let's say you want to parse an **HAL manifest** XML file using `xsdc`.

1. 📄 Write the XSD file

`hal_manifest.xsd` defines how your XML should look.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="manifest">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="hal" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence/>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="format" type="xs:string"/>
            <xs:attribute name="optional" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

2. 🧱 Define Android.bp

```bp
xsd_config {
    name: "hal_manifest",
    srcs: ["hal_manifest.xsd"],
    package_name: "hal.manifest",
}
```

This tells the build system:

> "Hey, generate Java & C++ code from `hal_manifest.xsd` under the package `hal.manifest`."

3. 📂 Directory Structure

```markdown
├── hal_manifest.xsd
├── Android.bp
└── api/
    ├── current.txt
    ├── last_current.txt
    ├── removed.txt
    └── last_removed.txt
```

You must run:

```bash
m update-api
```

To auto-fill these `api/*.txt` files.

🧪 How to Use the Code

🟡 Java Code

```java
import hal.manifest.*;

void readManifest(File file) {
    InputStream stream = new FileInputStream(file);
    Manifest manifest = XmlParser.read(stream);

    for (Hal hal : manifest.getHal()) {
        System.out.println("Name: " + hal.getName());
        System.out.println("Format: " + hal.getFormat());
        System.out.println("Optional: " + hal.getOptional());
    }
}
```

🔵 C++ Code

```cpp
#include "hal_manifest.h"  // generated

using namespace hal::manifest;

void readManifest(const std::string& fileName) {
    auto parsed = read(fileName.c_str());
    if (!parsed.has_value()) return;

    Manifest manifest = *parsed;
    for (auto& hal : manifest.getHal()) {
        std::cout << "Name: " << hal.getName() << "\n";
        std::cout << "Format: " << hal.getFormat() << "\n";
        std::cout << "Optional: " << hal.getOptional() << "\n";
    }
}
```

✅ What You Gained

Feature	Benefit
`xsdc` tool	Generates safe code from XML Schema
`xsd_config` rule	Easy to build and track schema like an API
`api/current.txt`	Prevents breaking schema accidentally
Java/C++ compatibility	Use same XML logic in both worlds
Schema → Objects	Read XML as Java/C++ classes, not text parsing

🧠 Summary in 1 Line:

> Use `xsdc` and `xsd_config` to **turn an XML schema (XSD)** into **strongly typed C++/Java code**, making XML parsing safer, cleaner, and backward-compatible.

Would you like a real minimal working example with dummy XML + XSD + code?

7. ↪
api/ ├── current.txt ├── last_current.txt ├── removed.txt └── last_removed.txt
what is this explain with example

Great question! Let's break down what these files in the `api/` folder mean, in **simple terms** — especially in the context of using `xsdc` for **Config File Schema APIs** in Android.

🧾 What Are These Files?

When you define **Java/C++ classes from XML schemas** (via `.xsd` files using `xsdc`), Android treats those class definitions **like public APIs** — just like the Java SDK.

To make sure:

- **You don't accidentally break** those APIs.
- You can track **backward-incompatible changes** (which would break existing vendor/system integration)...

Android's build system uses **API tracking files**:

File Name	Meaning
`current.txt`	📌 Snapshot of the **current API** (auto-generated from your XSD).
`last_current.txt`	📜 Snapshot of the **previous version's API** (used to check backward compatibility).
`removed.txt`	❌ List of **APIs you've removed** in the current version.
`last_removed.txt`	🧾 List of **previously removed APIs** (historical removals).

🧪 Example

Suppose you define this XML schema for audio config in `audio_config.xsd`:

```xml
<xs:element name="audioConfig">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="volume" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

You declare it in `Android.bp`:

```bp
xsd_config {
    name: "audio_config_parser",
    srcs: ["audio_config.xsd"],
    package_name: "audio.config",
}
```

When you run:

```bash
m update-api
```

It:

- **Parses** `audio_config.xsd`
- **Generates** Java/C++ classes like `AudioConfig` with a method `getVolume()`
- **Writes** the current API (e.g. `AudioConfig.getVolume()`) into:

```pgsql
api/
├── current.txt
├── last_current.txt
├── removed.txt
└── last_removed.txt
```

🛡️ Why Is This Helpful?

Let's say tomorrow you **change the schema** and **remove `volume`**. The generated Java class will **no longer have `getVolume()`**, and:

- Build system will detect this!
- It'll compare the new **`current.txt`** with **`last_current.txt`**
- It will **fail the build** or warn you: "Hey, you just broke an API!"

So these files **act like a version control system for your XML API**.

📌 Summary

File	Purpose
`current.txt`	What API looks like **right now**
`last_current.txt`	What it looked like **in the last version**
`removed.txt`	What you've **removed recently**
`last_removed.txt`	What was removed **in the past**

Let me know if you want to see what these files actually look like with some real content!

8. ConfigStore HAL

bookmark_border
In Android 10, ConfigStore HAL uses build flags to store config values in the vendor partition, and a service in the system partition accesses those values using HIDL (this is also true in Android 9). However, due to high memory consumption and difficult usage, the ConfigStore HAL has been deprecated.

The ConfigStore HAL remains in AOSP to support legacy vendor partitions. On devices running Android 10 or later, surfaceflinger reads system properties first; if no system property is defined for a config item in `SurfaceFlingerProperties.sysprop`, `surfaceflinger` falls back to the ConfigStore HAL.

Warning: Android 10 deprecates the ConfigStore HAL and replaces the HAL with system properties. For details, refer to Configuring.
Android 8.0 splits the monolithic Android OS into generic (system.img) and hardware-specific (vendor.img and odm.img) partitions. As a result of this change, conditional compilation must be removed from modules installed to the system partition and such modules must determine the configuration of the system at runtime (and behave differently depending on that configuration).

The ConfigStore HAL provides a set of APIs for accessing read-only configuration items used to configure the Android framework. This page describes the design of ConfigStore HAL (and why system properties weren't used for this purpose); other pages in this section detail the HAL interface, service implementation, and client-side usage, all using surfaceflinger as an example. For help with ConfigStore interface classes, see Adding Interface Classes and Items.

Why not use system properties?
We considered using system properties but found several fundamental issues, including:

Length limits on values. System properties have tight limits on the length of their values (92 bytes). In addition, as these limits have been directly exposed to Android apps as C macros, increasing the length can cause backward-compatibility issues.
No type support. All values are essentially strings, and APIs simply parse the string into an int or bool. Other compound data types (for example, array and struct) should be encoded/decoded by the clients (for example, "aaa,bbb,ccc" can be coded as an array of three strings).
Overwrites. Because read-only system properties are implemented as write-once properties, vendors/ODMs that want to override AOSP-defined read-only values must import their own read-only values prior to AOSP-defined read-only values. This, in turn, results in vendor-defined rewritable values being overridden by AOSP-defined values.
Address space requirements. System properties take a relatively large amount of address space in each process. System properties are grouped in prop_area units with a fixed size of 128 KB, all of which is allocated to a process address space even if only a single system property in it is being accessed. This can cause problems on 32-bit devices where address space is precious.
We attempted to overcome these limitations without sacrificing compatibility, but continued to be concerned that system properties weren't designed to support accessing read-only configuration items. Eventually we decided that system properties are better suited for sharing a few dynamically updated items across all of Android in real time, and that a need existed for a new system dedicated to accessing read-only configuration items.

ConfigStore HAL design
The basic design is simple:

Configstore HAL design

Figure 1. ConfigStore HAL design

Describe build flags (currently used for conditionally compiling the framework) in HIDL.
Vendors and OEMs provide SoC and device-specific values for build flags by implementing the HAL service.
Modify the framework to use the HAL service to find the value of a configuration item at runtime.
Configuration items currently referenced by the framework are included in a versioned HIDL package (android.hardware.configstore@1.0).
Vendors/OEMs provide values to the configuration items by implementing interfaces in this package, and the framework uses the interfaces when it needs to get a value for a configuration item.

Security considerations
Build flags defined in the same interface are affected by same SELinux policy. If one or more build flags should have different SELinux policies, they must be separated to another interface. This can require major revision of android.hardware.configstore package as the separated interfaces are no longer backward-compatible.

Note: For details on SELinux, see SELinux Overview.

Let me explain **ConfigStore HAL** in simple words with a real-world analogy and examples:

🔧 What is ConfigStore HAL?

ConfigStore HAL was a special **Hardware Abstraction Layer (HAL)** introduced before Android 10 that allowed the **framework (system partition)** to **read configuration values** (like display size, use of a specific GPU feature, etc.) provided by **vendors (vendor partition) at runtime**.

📦 Why ConfigStore HAL was needed?

In Android 8.0 and later, Android split into:

- **`system.img`**: generic Google code (e.g. SurfaceFlinger)
- **`vendor.img`**: hardware-specific code from OEMs

But the **`system.img`** needed to **know device-specific settings** (which live in the vendor partition). That created a problem — how do you let the system code **access vendor settings at runtime**?

So, Google introduced ConfigStore HAL — a kind of **read-only config service**.

📉 Why It Was Deprecated?

Google replaced it in Android 10+ because:

Problem with ConfigStore	Why it's bad
❗ High memory use	Every process using it had to load the full HAL service
❗ Complex to use	Needed to define HIDL, services, and link everything
❗ Overkill for simple values	Just to read "bool use_gpu=True" you had to define a full HAL
❗ Can't support fine-grained SELinux	You can't give different permissions to individual values easily
❗ Android wanted system properties instead	Properties are easier to manage, lighter, and integrate better

Now, ConfigStore is **deprecated**. It remains only to support old devices (Android 9 and earlier), but Android 10+ uses **system properties** instead.

🧪 Example (Simple Comparison)

🎯 Problem:

Let's say you want to let **`SurfaceFlinger`** know whether to use **wide-color** display feature for a specific hardware.

✅ Old method (Android 8–9 using ConfigStore HAL):

1. Define in HIDL:

```aidl
interface IDisplay {
    getWideColorSupport() generates (bool supported);
}
```

2. Vendor implements this HAL:

```cpp
bool IDisplay::getWideColorSupport() {
    return true;  // Only certain devices
}
```

3. **`SurfaceFlinger`** framework calls:

```cpp
auto supported = configstore::ISurfaceFlingerConfigs::getWideColorSupport();
```

> ✨ Works fine, but complex. Needs HIDL setup, service registration, SEPolicy, etc.

✅ New method (Android 10+ using system property):

Vendor sets this in **`vendor.prop`**:

```properties
persist.vendor.display.wide_color_support=true
```

`SurfaceFlinger` reads it with:

```cpp
bool supported = base::GetBoolProperty("persist.vendor.display.wide_color_support", false);
```

> ✨ Much simpler: no HAL, no IPC, no memory overhead.

🔒 Security Implications

- In ConfigStore, **all config items under the same interface** share the **same SELinux policy**.
- If some configs need tighter access control, they'd have to move to **different HAL interfaces**, which is hard to manage and not backward-compatible.

This inflexibility was another reason for deprecation.

📝 Summary

Aspect	ConfigStore HAL	System Properties
Introduced	Android 8.0	Always existed
Use case	Runtime read-only config from vendor	Runtime config, both static and dynamic
Flexible typing	Yes (bool, int, etc.)	No (all strings)
Secure	SELinux tied to interface	Fine-grained (via prop_contexts)
Memory	High usage	Low footprint
Current status	**Deprecated in Android 10**	Preferred

✅ Final Takeaway

> **Before Android 10:** Vendors used ConfigStore HAL to send read-only configs to system code at runtime.
>
> **After Android 10:** Use **system properties** (**`ro.vendor.`**, **`persist.vendor.`**) instead — they are lighter, faster, and easier to manage.

Let me know if you'd like a working code snippet or a minimal AOSP example!

9. ConfigStore

bookmark_border
In Android, system properties are stored in the ConfigStore class. ConfigStore uses build flags to store config values in the vendor partition, and a service in the system partition accesses those values using HIDL.

System properties use PRODUCT_DEFAULT_PROPERTY_OVERRIDES to store system properties in default.prop in the vendor partition, and the service uses sysprop to read those properties.
The ConfigStore HAL remains in AOSP to support legacy vendor partitions. On devices running Android 10, surfaceflinger reads system properties first; if no system property is defined for a config item in SurfaceFlingerProperties.sysprop, surfaceflinger falls back to the ConfigStore HAL.

Build flags and system properties
Each build flag in ConfigStore has a matching system property, as shown in the following table.

Build flags	System properties
TARGET_FORCE_HWC_FOR_VIRTUAL_DISPLAYS	ro.surface_flinger.force_hwc_copy_for_virtual_displays
TARGET_HAS_HDR_DISPLAY	ro.surface_flinger.has_HDR_display
TARGET_HAS_WIDE_COLOR_DISPLAY	ro.surface_flinger.has_wide_color_display
NUM_FRAMEBUFFER_SURFACE_BUFFERS	ro.surface_flinger.max_frame_buffer_acquired_buffers
MAX_VIRTUAL_DISPLAY_DIMENSION	ro.surface_flinger.max_virtual_display_dimension
PRIMARY_DISPLAY_ORIENTATION	ro.surface_flinger.primary_display_orientation
PRESENT_TIME_OFFSET_FROM_VSYNC_NS	ro.surface_flinger.present_time_offset_from_vsync_ns
TARGET_RUNNING_WITHOUT_SYNC_FRAMEWORK	ro.surface_flinger.running_without_sync_framework
SF_START_GRAPHICS_ALLOCATOR_SERVICE	ro.surface_flinger.start_graphics_allocator_service
TARGET_USE_CONTEXT_PRIORITY	ro.surface_flinger.use_context_priority
USE_VR_FLINGER	ro.surface_flinger.use_vr_flinger
VSYNC_EVENT_PHASE_OFFSET_NS	ro.surface_flinger.vsync_event_phase_offset_ns
SF_VSYNC_EVENT_PHASE_OFFSET_NS	ro.surface_flinger.vsync_sf_event_phase_offset_ns

New system properties
Android 10 includes the following new system properties:

ro.surface_flinger.default_composition_dataspace
ro.surface_flinger.default_composition_pixel_format
ro.surface_flinger.use_color_management
ro.surface_flinger.wcg_composition_dataspace
ro.surface_flinger.wcg_composition_pixel_format
ro.surface_flinger.display_primary_red
ro.surface_flinger.display_primary_green
ro.surface_flinger.display_primary_blue
ro.surface_flinger.display_primary_white
ro.surface_flinger.protected_contents
ro.surface_flinger.set_idle_timer_ms
ro.surface_flinger.set_touch_timer_ms
ro.surface_flinger.use_smart_90_for_video
ro.surface_flinger.protected_contents
ro.surface_flinger.support_kernel_idle_timer
For details on these properties, refer to frameworks/native/services/surfaceflinger/sysprop/SurfaceFlingerProperties.sysprop.

Use SurfaceFlingerProperties
In the following examples for the SurfaceFlingerProperties library, the function name is api_name in SurfaceFlingerProperties.sysprop.

```
cc_binary {
    name: "cc_client",
    srcs: ["baz.cpp"],
    shared_libs: ["SurfaceFlingerProperties"],
}
java_library {
    name: "JavaClient",
    srcs: ["foo/bar.java"],
    libs: ["SurfaceFlingerProperties"],
}
```

```
import android.sysprop.SurfaceFlingerProperties;
...

static void foo() {
    ...
    boolean temp = SurfaceFlingerProperties.vsync_event_phase_offset_ns().orElse(true);
    ...
}
...

#include <SurfaceFlingerProperties.sysprop.h>
using namespace android::sysprop;

...

void bar() {
    ...
    bool temp = SurfaceFlingerProperties::vsync_event_phase_offset_ns(true);
    ...
}
...
```

Here's a **simple explanation** of the `ConfigStore` system with **real-life examples** and why it's being replaced by `SurfaceFlingerProperties.sysprop` and system properties in Android 10+.

🧠 What is ConfigStore?

Think of **ConfigStore** as a **read-only configuration dictionary** that helps Android system services (like `SurfaceFlinger`) know about **hardware-specific features** — such as:

- Does the device support HDR?
- What's the display orientation?
- Should SurfaceFlinger use VR rendering?

💡 Real-Life Analogy

Imagine you're building an Android TV, and the system (like SurfaceFlinger) wants to know:

> "Does this device support wide color display?"

Before Android 10, the system asked the **ConfigStore HAL**:

> “Hey ConfigStore, does this device have wide color?”

ConfigStore would answer:

> “Yes. Because I was built with `TARGET_HAS_WIDE_COLOR_DISPLAY := true`.”

🏗️ How it worked (Android 9 and earlier)

1. Vendor set **build flags** in `BoardConfig.mk`, like:

```make
TARGET_HAS_WIDE_COLOR_DISPLAY := true
```

2. These flags were turned into **HIDL HAL implementations** (in ConfigStore).
3. System services (like SurfaceFlinger) accessed values using **HIDL calls**.

📉 Why ConfigStore Was Deprecated in Android 10+

Problems with ConfigStore:

Problem	Why it was bad
🧠 Complex to implement	Needed HIDL service per flag
🐏 Consumed memory	Even unused HALs consumed space
⚠️ Limited flexibility	Difficult to update or override
🔐 SELinux problems	All flags in one interface shared the same security policy

✅ New Way: Use `SurfaceFlingerProperties.sysprop`

Instead of build flags and HIDL, Android 10+ uses **system properties** (defined at build time) and a **generated C++/Java API** to access them.

🧾 Example Mapping

Old Build Flag	New System Property
`TARGET_HAS_WIDE_COLOR_DISPLAY`	`ro.surface_flinger.has_wide_color_display`
`TARGET_USE_CONTEXT_PRIORITY`	`ro.surface_flinger.use_context_priority`
`USE_VR_FLINGER`	`ro.surface_flinger.use_vr_flinger`

These values now live in `build.prop` and are read using system property APIs.

🔍 Example in Java

```java
import android.sysprop.SurfaceFlingerProperties;

boolean isWideColorSupported =
    SurfaceFlingerProperties.has_wide_color_display().orElse(false);
```

> ✅ Automatically reads system property `ro.surface_flinger.has_wide_color_display`

🧑‍💻 Example in C++

```cpp
#include <SurfaceFlingerProperties.sysprop.h>

bool isWideColorSupported =
    android::sysprop::SurfaceFlingerProperties::has_wide_color_display().value_or(false);
```

> ✅ Reads system property and returns a typed result (`bool`, `int`, etc.)

📦 How to Set These Properties

In `device.mk` or `BoardConfig.mk`, add:

```make
PRODUCT_SYSTEM_PROPERTIES += ro.surface_flinger.has_wide_color_display=true
```

Or create a prop file:

```make
TARGET_VENDOR_PROP += vendor/display.prop
```

With contents:

```properties
ro.surface_flinger.has_HDR_display=true
```

🧩 Where Are These Defined?

All the new SurfaceFlinger properties are defined in:

```swift
frameworks/native/services/surfaceflinger/sysprop/SurfaceFlingerProperties.sysprop
```

This `.sysprop` file:

- Defines the system property names
- Maps them to typed getters
- Can generate Java/C++/Rust APIs automatically

🧠 Summary

Before (Android 9 and earlier)	After (Android 10+)
Used **`ConfigStore`** HAL (via HIDL)	Uses system properties
Values hardcoded via build flags	Values set via **`.prop`** or makefiles
Accessed using HIDL services	Accessed using **`SurfaceFlingerProperties`** APIs
Complex, heavy, deprecated	Light, simple, modern

✅ Final Takeaway

> Android 10 **replaces** ConfigStore HAL with **system properties** + **auto-generated typed APIs** like **`SurfaceFlingerProperties`**.

It's simpler, faster, safer, and easier to maintain.

Let me know if you want a full example **`.sysprop`** file + Java/C++ usage together!

10. Create the HAL interface

bookmark_border
Warning: Android 10 deprecates the ConfigStore HAL and replaces the HAL with system properties. For details, refer to Configuring.
You must use HIDL to describe all build flags used for conditionally compiling the framework. Relevant build flags must be grouped and included in a single .hal file. Using HIDL for specifying configuration items includes the following benefits:

Versioned (to add new config items, vendors/OEMs must explicitly extend the HAL)
Well-documented
Access control using SELinux
Sanity check for configuration items through the Vendor Test Suite (range check, inter-dependency check among items, etc.)
Auto-generated APIs in both C++ and Java
Identify build flags used by the framework
Start by identifying the build configs used to conditionally compile the framework, then abandon obsolete configs to make the set smaller. For example, the following set of build flags are identified for surfaceflinger:

TARGET_USES_HWC2
TARGET_BOARD_PLATFORM
TARGET_DISABLE_TRIPLE_BUFFERING
TARGET_FORCE_HWC_FOR_VIRTUAL_DISPLAYS
NUM_FRAMEBUFFER_SURFACE_BUFFERS
TARGET_RUNNING_WITHOUT_SYNC_FRAMEWORK
VSYNC_EVENT_PHASE_OFFSET_NS
SF_VSYNC_EVENT_PHASE_OFFSET_NS
PRESENT_TIME_OFFSET_FROM_VSYNC_NS
MAX_VIRTUAL_DISPLAY_DIMENSION
Create a HAL interface
Build configs for a subsystem are accessed through a HAL interface, while interfaces for giving configuration values are grouped in the HAL package android.hardware.configstore (currently at version 1.0). For example, to create a HAL interface file for surfaceflinger, in hardware/interfaces/configstore/1.0/ISurfaceFlingerConfigs.hal:

```
package android.hardware.configstore@1.0;

interface ISurfaceFlingerConfigs {
    // TO-BE-FILLED-BELOW
};
```
After creating the .hal file, run hardware/interfaces/update-makefiles.sh to add the new .hal file to the Android.bp and Android.mk files.

Add functions for build flags
For each build flag, add a new function to the interface. For example, in hardware/interfaces/configstore/1.0/ISurfaceFlingerConfigs.hal:

```
interface ISurfaceFlingerConfigs {
    disableTripleBuffering() generates(OptionalBool ret);
    forceHwcForVirtualDisplays() generates(OptionalBool ret);
    enum NumBuffers: uint8_t {
        USE_DEFAULT = 0,
        TWO = 2,
        THREE = 3,
    };
    numFramebufferSurfaceBuffers() generates(NumBuffers ret);
    runWithoutSyncFramework() generates(OptionalBool ret);
    vsyncEventPhaseOffsetNs generates (OptionalUInt64 ret);
    presentTimeOffsetFromSyncNs generates (OptionalUInt64 ret);
    maxVirtualDisplayDimension() generates(OptionalInt32 ret);
};
```
When adding a function:

Be concise with names. Avoid converting makefile variable names into function names and keep in mind that TARGET_ and BOARD_ prefixes are no longer necessary.
Add comments. Help developers understand the purpose of the config item, how it changes framework behavior, valid values, and other relevant information.

Function return types can be Optional[Bool|String|Int32|UInt32|Int64|UInt64]. Types are defined in types.hal in the same directory and wrap primitive values with a field that indicates if the value is specified by the HAL; if not, the default value is used.

```
struct OptionalString {
   bool specified;
   string value;
};
```

When appropriate, define the enum that best represents the type of the configuration item and use that enum as the return type. In the example above, the NumBuffers enum is defined to limit the number of valid values. When defining such custom data types, add a field or a enum value (for example, USE_DEFAULT) for denoting if the value is/isn't specified by the HAL.

It's not mandatory for a single build flag to become a single function in HIDL. Module owners can alternatively aggregate closely related build flags into a struct and have a function that returns that struct (doing so can reduce number of function calls).

For example, an option for aggregating two build flags into a single struct in hardware/interfaces/configstore/1.0/ISurfaceFlingerConfigs.hal is:

```
 interface ISurfaceFlingerConfigs {
   // other functions here
   struct SyncConfigs {
      OptionalInt64 vsyncEventPhaseoffsetNs;
      OptionalInt64 presentTimeoffsetFromSyncNs;
   };
   getSyncConfigs() generates (SyncConfigs ret);
   // other functions here
};
```

Alternatives to a single HAL function
As an alternative to using a single HAL function for all build flags, the HAL interface also provides simple functions such as getBoolean(string key) and getInteger(string key). The actual key=value pairs are stored in separate files and the HAL service provides values by reading/parsing those files.

While this approach is easy to define, it doesn't include the benefits provided by HIDL (enforced versioning, ease of documentation, access control) and is therefore not recommended.

Note: When using simple functions, access control is extremely difficult as the HAL cannot identify clients by itself.
Single and multiple interfaces
The design of the HAL interface for configuration items presents two choices:

A single interface that covers all configuration items
Multiple interfaces, each of which covers a set of related configuration items
A single interface is easier but can become unmaintainable as more configuration items are added to the single file. In addition, access control isn't fine-grained, so a process that's granted access to the interface can read all configuration items (access to a partial set of configuration items can't be granted). Alternatively, if access isn't granted, configuration items can't be read.

Because of these issues, Android uses multiple interfaces with a single HAL interface for a group of related configuration items. For example, ISurfaceflingerConfigs for surfaceflinger-related configuration items, and IBluetoothConfigs for Bluetooth-related configuration items.

Here's a **simple explanation** of how to **create a HAL interface for ConfigStore**, why it was done this way before Android 10, and how to actually define such an interface using HIDL — with clear examples.

🌟 What's the Goal?

In older Android versions (before Android 10), many system components like **SurfaceFlinger**, **Bluetooth**, etc., needed access to **hardware-specific configuration values** (like number of framebuffers, or if HDR is supported).

So, a **ConfigStore HAL** was created to expose these values in a **standard, secure, and versioned way** — using **HIDL interfaces**.

> ⚠️ Note: Android 10 and later **deprecated** this in favor of **system properties** like `ro.surface_flinger.xxx`.

🏗️ What is a HAL Interface?

A **HAL interface** is a `.hal` file written using **HIDL syntax** that defines:

- **What configuration values exist** (as functions)

- **What types they return** (booleans, integers, enums, etc.)
- **How the system will ask for those values at runtime**

✅ Step-by-Step Example: Create a ConfigStore HAL for SurfaceFlinger

---

🔍 Step 1: Identify Build Flags Used

Imagine you found the following build flags used for **`SurfaceFlinger`**:

```make
TARGET_DISABLE_TRIPLE_BUFFERING := true
TARGET_FORCE_HWC_FOR_VIRTUAL_DISPLAYS := true
NUM_FRAMEBUFFER_SURFACE_BUFFERS := 3
VSYNC_EVENT_PHASE_OFFSET_NS := 5000000
```

📁 Step 2: Create a `.hal` Interface File

Create this file:

```swift
hardware/interfaces/configstore/1.0/ISurfaceFlingerConfigs.hal
```

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

interface ISurfaceFlingerConfigs {
    // Whether triple buffering is disabled
    disableTripleBuffering() generates (OptionalBool ret);

    // Whether hardware composer (HWC) is forced for virtual displays
    forceHwcForVirtualDisplays() generates (OptionalBool ret);

    // Number of framebuffer surface buffers
    enum NumBuffers: uint8_t {
        USE_DEFAULT = 0,
        TWO = 2,
        THREE = 3,
    };
    numFramebufferSurfaceBuffers() generates (NumBuffers ret);

    // Vsync phase offset in nanoseconds
    vsyncEventPhaseOffsetNs() generates (OptionalUInt64 ret);
};
```

📚 Optional Types Used

These come from **`types.hal`**:

```hidl
struct OptionalBool {
    bool specified;
    bool value;
};

struct OptionalUInt64 {
    bool specified;
    uint64_t value;
};
```

> 🔍 **`specified`** tells the system if the value was actually defined by the vendor.

🧠 Why Use Enums?

Enums like this:

```hidl
enum NumBuffers: uint8_t {
    USE_DEFAULT = 0,
    TWO = 2,
    THREE = 3,
};
```

are used instead of raw integers to **limit** allowed values and **self-document** what each value means.

🔃 Step 3: Update Build Files

Run the following script to auto-generate the necessary makefiles:

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

This updates `Android.bp` and `Android.mk`.

💡 Step 4: Implement the HAL

Now, you write the actual service that implements this interface — usually in:

```swift
hardware/interfaces/configstore/1.0/default/
```

Example C++ implementation:

```cpp
Return<void> ISurfaceFlingerConfigs::disableTripleBuffering(disableTripleBuffering_cb _hidl_cb) {
    OptionalBool ret;
    ret.specified = true;
    ret.value = true; // Or false depending on build config
    _hidl_cb(ret);
    return Void();
}
```

🧑‍💻 Step 5: Access from System

System components (e.g., SurfaceFlinger) access the HAL like this (in C++):

```cpp
sp<ISurfaceFlingerConfigs> config = ISurfaceFlingerConfigs::getService();
OptionalBool result = config->disableTripleBuffering();
if (result.specified && result.value) {
    // Use triple buffering
}
```

🔁 Alternative: Aggregate Related Configs in Structs

To reduce function calls, you can return multiple related values in one struct:

```hidl
interface ISurfaceFlingerConfigs {
    struct SyncConfigs {
        OptionalUInt64 vsyncEventPhaseOffsetNs;
        OptionalUInt64 presentTimeOffsetFromSyncNs;
    };

    getSyncConfigs() generates (SyncConfigs ret);
};
```

🚫 Why This Was Deprecated

Although powerful, this system was:

Issue	Reason
🧠 Complex	Required HIDL service per config group
🔐 Rigid	SELinux policies not fine-grained enough
🐏 Memory Heavy	Needed HALs always in memory
❌ Not Scalable	Too many HALs for simple values

✅ Android 10+ Solution: Use System Properties

Instead of a HAL function, just write:

```make
PRODUCT_SYSTEM_PROPERTIES += ro.surface_flinger.use_context_priority=true
```

And access it in code:

```cpp
#include <SurfaceFlingerProperties.sysprop.h>
bool usePriority = SurfaceFlingerProperties::use_context_priority().value_or(false);
```

🧠 Summary

Before (Android 9 and earlier)	After (Android 10+)
Use HIDL-based ConfigStore HAL	Use system properties with sysprop APIs
Required `.hal` files and C++/Java HAL services	Use `.sysprop` file + typed APIs
Complicated, but versioned	Simpler and easier to maintain

Would you like a **real .hal + .cpp + sysprop example project structure** next?