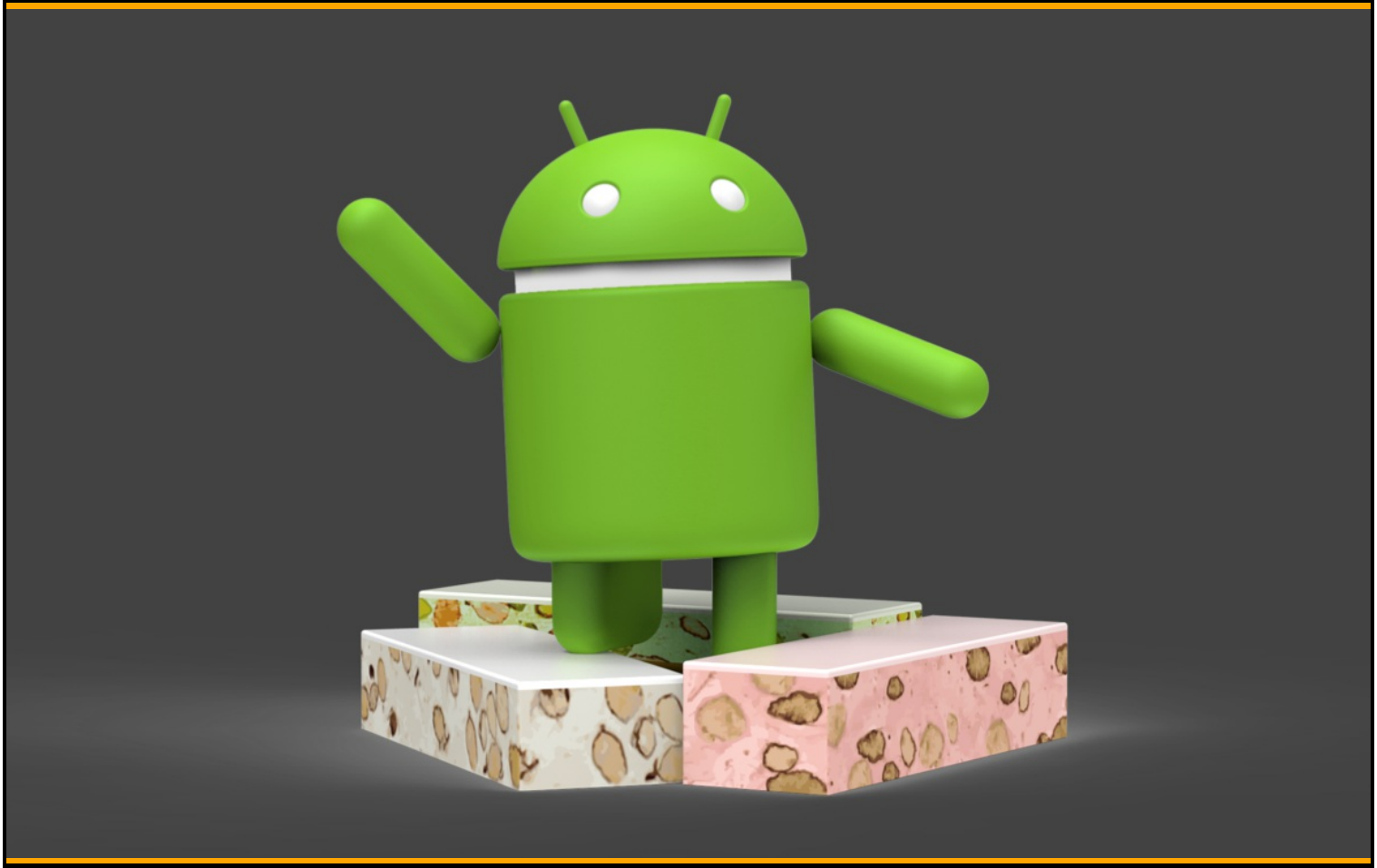
android

Compatibility Definition



Android 7.1

Last updated: December 20th, 2016

compatibility@android.com

Table of Contents

1. Introduction

This document enumerates the requirements that must be met in order for devices to be compatible with Android 7.1.

The use of “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” is per the IETF standard defined in RFC2119.

As used in this document, a “device implementer” or “implementer” is a person or organization developing a hardware/software solution running Android 7.1. A “device implementation” or “implementation is the hardware/software solution so developed.

To be considered compatible with Android 7.1, device implementations MUST meet the requirements presented in this Compatibility Definition, including any documents incorporated via reference.

Where this definition or the software tests described in section 10 is silent, ambiguous, or incomplete, it is the responsibility of the device implementer to ensure compatibility with existing implementations.

For this reason, the Android Open Source Project is both the reference and preferred implementation of Android. Device implementers are STRONGLY RECOMMENDED to base their implementations to the greatest extent possible on the “upstream” source code available from the Android Open Source Project. While some components can hypothetically be replaced with alternate implementations, it is STRONGLY RECOMMENDED to not follow this practice, as passing the software tests will become substantially more difficult. It is the implementer’s responsibility to ensure full behavioral compatibility with the standard Android implementation, including and beyond the Compatibility Test Suite. Finally, note that certain component substitutions and modifications are explicitly forbidden by this document.

Many of the resources linked to in this document are derived directly or indirectly from the Android SDK and will be functionally identical to the information in that SDK’s documentation. In any cases where this Compatibility Definition or the Compatibility Test Suite disagrees with the SDK documentation, the SDK documentation is considered authoritative. Any technical details provided in the linked resources throughout this document are considered by inclusion to be part of this Compatibility Definition.

2. Device Types

While the Android Open Source Project has been used in the implementation of a variety of device types and form factors, many aspects of the architecture and compatibility requirements were optimized for handheld devices. Starting from Android 5.0, the Android Open Source Project aims to embrace a wider variety of device types as described in this section.

Android Handheld device refers to an Android device implementation that is typically used by holding it in the hand, such as mp3 players, phones, and tablets. Android Handheld device implementations:

- MUST have a touchscreen embedded in the device.
- MUST have a power source that provides mobility, such as a battery.

Android Television device refers to an Android device implementation that is an entertainment interface for consuming digital media, movies, games, apps, and/or live TV for users sitting about ten feet away (a “lean back” or “10-foot user interface”). Android Television devices:

- MUST have an embedded screen OR include a video output port, such as VGA, HDMI, or a wireless port for display.
- MUST declare the features android.software.leanback and android.hardware.type.television.

Android Watch device refers to an Android device implementation intended to be worn on the body, perhaps on the wrist, and:

- MUST have a screen with the physical diagonal length in the range from 1.1 to 2.5 inches.
- MUST declare the feature android.hardware.type.watch.
- MUST support uiMode = UI_MODE_TYPE_WATCH.

Android Automotive implementation refers to a vehicle head unit running Android as an operating system for part or all of the system and/or infotainment functionality. Android Automotive implementations:

- MUST have a screen with the physical diagonal length equal to or greater than 6 inches.
- MUST declare the feature android.hardware.type.automotive.
- MUST support uiMode = UI_MODE_TYPE_CAR.
- Android Automotive implementations MUST support all public APIs in the android.car.* namespace.

All Android device implementations that do not fit into any of the above device types still MUST meet all requirements in this document to be Android 7.1 compatible, unless the requirement is explicitly described to be only applicable to a specific Android device type from above.

2.1 Device Configurations

This is a summary of major differences in hardware configuration by device type. (Empty cells denote a “MAY”). Not all configurations are covered in this table; see relevant hardware sections for more detail.

Category	Feature	Section	Handheld	Television	Watch	Automotive	Other
Input	D-pad	7.2.2. Non-touch Navigation		MUST			
	Touchscreen	7.2.4. Touchscreen input	MUST		MUST		SHOULD
	Microphone	7.8.1. Microphone	MUST	SHOULD	MUST	MUST	SHOULD
Sensors	Accelerometer	7.3.1 Accelerometer	SHOULD		SHOULD		SHOULD
	GPS	7.3.3. GPS	SHOULD			SHOULD	
Connectivity	Wi-Fi	7.4.2. IEEE 802.11	SHOULD	SHOULD		SHOULD	SHOULD
	Wi-Fi Direct	7.4.2.1. Wi-Fi Direct	SHOULD	SHOULD			SHOULD
	Bluetooth	7.4.3. Bluetooth	SHOULD	MUST	MUST	MUST	SHOULD
	Bluetooth Low Energy	7.4.3. Bluetooth	SHOULD	MUST	SHOULD	SHOULD	SHOULD
	Cellular radio	7.4.5. Minimum Network Capability				SHOULD	
USB							

peripheral/host mode	7.7. USB	SHOULD			SHOULD	SHOULD	
Output	Speaker and/or Audio output ports	7.8.2. Audio Output	MUST	MUST		MUST	MUST

3. Software

3.1. Managed API Compatibility

The managed Dalvik bytecode execution environment is the primary vehicle for Android applications. The Android application programming interface (API) is the set of Android platform interfaces exposed to applications running in the managed runtime environment. Device implementations MUST provide complete implementations, including all documented behaviors, of any documented API exposed by the Android SDK or any API decorated with the "@SystemApi" marker in the upstream Android source code.

Device implementations MUST support/preserve all classes, methods, and associated elements marked by the TestApi annotation (@TestApi).

Device implementations MUST NOT omit any managed APIs, alter API interfaces or signatures, deviate from the documented behavior, or include no-ops, except where specifically allowed by this Compatibility Definition.

This Compatibility Definition permits some types of hardware for which Android includes APIs to be omitted by device implementations. In such cases, the APIs MUST still be present and behave in a reasonable way. See section 7 for specific requirements for this scenario.

3.1.1. Android Extensions

Android includes the support of extending the managed APIs while keeping the same API level version. Android device implementations MUST preload the AOSP implementation of both the shared library ExtShared and services ExtServices with versions higher than or equal to the minimum versions allowed per each API level. For example, Android 7.0 device implementations, running API level 24 MUST include at least version 1.

3.2. Soft API Compatibility

In addition to the managed APIs from section 3.1, Android also includes a significant runtime-only "soft" API, in the form of such things as intents, permissions, and similar aspects of Android applications that cannot be enforced at application compile time.

3.2.1. Permissions

Device implementers MUST support and enforce all permission constants as documented by the Permission reference page. Note that section 9 lists additional requirements related to the Android security model.

3.2.2. Build Parameters

The Android APIs include a number of constants on the android.os.Buildclass that are intended to describe the current device. To provide consistent, meaningful values across device implementations, the table below includes additional restrictions on the formats of these values to which device implementations MUST conform.

Parameter	Details
VERSION.RELEASE	The version of the currently-executing Android system, in human-readable format. This field MUST have one of the string values defined in 7.1.
VERSION.SDK	The version of the currently-executing Android system, in a format accessible to third-party application code. For Android 7.1, this field MUST have the integer value 7.1_INT.
VERSION.SDK_INT	The version of the currently-executing Android system, in a format accessible to third-party application code. For Android 7.1, this field MUST have the integer value 7.1_INT.
VERSION.INCREMENTAL	A value chosen by the device implementer designating the specific build of the currently-executing Android system, in human-readable format. This value MUST NOT be reused for different builds made available to end users. A typical use of this field is to indicate which build number or source-control change identifier was used to generate the build. There are no requirements on the specific format of this field, except that it MUST NOT be null or the empty string ("").
BOARD	A value chosen by the device implementer identifying the specific internal hardware used by the device, in human-readable format. A possible use of this field is to indicate the specific revision of the board powering the device. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression “^[a-zA-Z0-9_-]+$”.
BRAND	A value reflecting the brand name associated with the device as known to the end users. MUST be in human-readable format and SHOULD represent the manufacturer of the device or the company brand under which the device is marketed. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression “^[a-zA-Z0-9_-]+$”.
SUPPORTED_ABIS	The name of the instruction set (CPU type + ABI convention) of native code. See section 3.3. Native APICompatibility.
SUPPORTED_32_BIT_ABIS	The name of the instruction set (CPU type + ABI convention) of native code. See section 3.3. Native APICompatibility.
SUPPORTED_64_BIT_ABIS	The name of the second instruction set (CPU type + ABI convention) of native code. See section 3.3. NativeAPI Compatibility.
CPU_ABI	The name of the instruction set (CPU type + ABI convention) of native code. See section 3.3. Native APICompatibility.
CPU_ABI2	The name of the second instruction set (CPU type + ABI convention) of native code. See section 3.3. NativeAPI Compatibility.
DEVICE	A value chosen by the device implementer containing the development name or code name identifying the configuration of the hardware features and industrial design of the device. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression “^[a-zA-Z0-9_-]+$”. This device name MUST NOT change during the lifetime of the product.
	A string that uniquely identifies this build. It SHOULD be reasonably human-readable. It MUST follow this template: **$(BRAND)/$(PRODUCT)/** **$(DEVICE):$(VERSION.RELEASE)/$(ID)/$(VERSION.INCREMENTAL):$(TYPE)/$(TAGS)**

FINGERPRINT	For example: **acme/myproduct/** **mydevice:7.1/LMYXX/3359:userdebug/test-keys** The fingerprint MUST NOT include whitespace characters. If other fields included in the template above have whitespace characters, they MUST be replaced in the build fingerprint with another character, such as the underscore ("_") character. The value of this field MUST be encodable as 7-bit ASCII.
HARDWARE	The name of the hardware (from the kernel command line or /proc). It SHOULD be reasonably human-readable. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression “^[a-zA-Z0-9_-]+$”.
HOST	A string that uniquely identifies the host the build was built on, in human-readable format. There are no requirements on the specific format of this field, except that it MUST NOT be null or the empty string ("").
ID	An identifier chosen by the device implementer to refer to a specific release, in human-readable format. This field can be the same as android.os.Build.VERSION.INCREMENTAL, but SHOULD be a value sufficiently meaningful for end users to distinguish between software builds. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression “^[a-zA-Z0-9._-]+$”.
MANUFACTURER	The trade name of the Original Equipment Manufacturer (OEM) of the product. There are no requirements on the specific format of this field, except that it MUST NOT be null or the empty string ("").
MODEL	A value chosen by the device implementer containing the name of the device as known to the end user. This SHOULD be the same name under which the device is marketed and sold to end users. There are no requirements on the specific format of this field, except that it MUST NOT be null or the empty string ("").
PRODUCT	A value chosen by the device implementer containing the development name or code name of the specific product (SKU) that MUST be unique within the same brand. MUST be human-readable, but is not necessarily intended for view by end users. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression “^[a-zA-Z0-9_-]+$”. This product name MUST NOT change during the lifetime of the product.
SERIAL	A hardware serial number, which MUST be available and unique across devices with the same MODEL and MANUFACTURER. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression “^([a-zA-Z0-9]{6,20})$”.
TAGS	A comma-separated list of tags chosen by the device implementer that further distinguishes the build. This field MUST have one of the values corresponding to the three typical Android platform signing configurations: release-keys, dev-keys, test-keys.
TIME	A value representing the timestamp of when the build occurred.
TYPE	A value chosen by the device implementer specifying the runtime configuration of the build. This field MUST have one of the values corresponding to the three typical Android runtime configurations: user, userdebug, or eng.

USER	A name or user ID of the user (or automated user) that generated the build. There are no requirements on the specific format of this field, except that it MUST NOT be null or the empty string ("").
SECURITY_PATCH	A value indicating the security patch level of a build. It MUST signify that the build includes all security patches issued up through the designated Android Public Security Bulletin. It MUST be in the format [YYYY-MM-DD], matching one of the Android Security Patch Level strings of the Public Security Bulletins, for example "2015-11-01".
BASE_OS	A value representing the FINGERPRINT parameter of the build that is otherwise identical to this build except for the patches provided in the Android Public Security Bulletin. It MUST report the correct value and if such a build does not exist, report an empty string ("").

3.2.3. Intent Compatibility

3.2.3.1. Core Application Intents

Android intents allow application components to request functionality from other Android components. The Android upstream project includes a list of applications considered core Android applications, which implements several intent patterns to perform common actions. The core Android applications are:

- Desk Clock
- Browser
- Calendar
- Contacts
- Gallery
- GlobalSearch
- Launcher
- Music
- Settings

Device implementations MUST include the core Android applications as appropriate or a component implementing the same intent patterns defined by all the Activity or Service components of these core Android applications exposed to other applications, implicitly or explicitly, through the android:exported attribute.

3.2.3.2. Intent Resolution

As Android is an extensible platform, device implementations MUST allow each intent pattern referenced in section 3.2.3.1 to be overridden by third-party applications. The upstream Android open source implementation allows this by default; device implementers MUST NOT attach special privileges to system applications' use of these intent patterns, or prevent third-party applications from binding to and assuming control of these patterns. This prohibition specifically includes but is not limited to disabling the “Chooser” user interface that allows the user to select between multiple applications that all handle the same intent pattern.

Device implementations MUST provide a user interface for users to modify the default activity for intents.

However, device implementations MAY provide default activities for specific URI patterns (e.g. http://play.google.com) when the default activity provides a more specific attribute for the data URI. For example, an intent filter pattern specifying the data URI “http://www.android.com” is more specific

than the browser's core intent pattern for “http://”.

Android also includes a mechanism for third-party apps to declare an authoritative default app linking behavior for certain types of web URI intents. When such authoritative declarations are defined in an app's intent filter patterns, device implementations:

- MUST attempt to validate any intent filters by performing the validation steps defined in the Digital Asset Links specification as implemented by the Package Manager in the upstream Android Open Source Project.
- MUST attempt validation of the intent filters during the installation of the application and set all successfully validated UIR intent filters as default app handlers for their UIRs.
- MAY set specific URI intent filters as default app handlers for their URIs, if they are successfully verified but other candidate URI filters fail verification. If a device implementation does this, it MUST provide the user appropriate per-URI pattern overrides in the settings menu.
- MUST provide the user with per-app App Links controls in Settings as follows:
  - The user MUST be able to override holistically the default app links behavior for an app to be: always open, always ask, or never open, which must apply to all candidate URI intent filters equally.
  - The user MUST be able to see a list of the candidate URI intent filters.
  - The device implementation MAY provide the user with the ability to override specific candidate URI intent filters that were successfully verified, on a per-intent filter basis.
  - The device implementation MUST provide users with the ability to view and override specific candidate URI intent filters if the device implementation lets some candidate URI intent filters succeed verification while some others can fail.

3.2.3.3. Intent Namespaces

Device implementations MUST NOT include any Android component that honors any new intent or broadcast intent patterns using an ACTION, CATEGORY, or other key string in the android. *or com.android.* namespace. Device implementers MUST NOT include any Android components that honor any new intent or broadcast intent patterns using an ACTION, CATEGORY, or other key string in a package space belonging to another organization. Device implementers MUST NOT alter or extend any of the intent patterns used by the core apps listed in section 3.2.3.1. Device implementations MAY include intent patterns using namespaces clearly and obviously associated with their own organization. This prohibition is analogous to that specified for Java language classes in section 3.6.

3.2.3.4. Broadcast Intents

Third-party applications rely on the platform to broadcast certain intents to notify them of changes in the hardware or software environment. Android-compatible devices MUST broadcast the public broadcast intents in response to appropriate system events. Broadcast intents are described in the SDK documentation.

3.2.3.5. Default App Settings

Android includes settings that provide users an easy way to select their default applications, for example for Home screen or SMS. Where it makes sense, device implementations MUST provide a similar settings menu and be compatible with the intent filter pattern and API methods described in the SDK documentation as below.

Device implementations:

- MUST honor the android.settings.HOME_SETTINGS intent to show a default app settings menu for Home Screen, if the device implementation reports android.software.home_screen.
- MUST provide a settings menu that will call the android.provider.Telephony.ACTION_CHANGE_DEFAULT intent to show a dialog to change the default SMS application, if the device implementation reports android.hardware.telephony.
- MUST honor the android.settings.NFC_PAYMENT_SETTINGS intent to show a default app settings menu for Tap and Pay, if the device implementation reports android.hardware.nfc.hce.
- MUST honor the android.telecom.action.CHANGE_DEFAULT_DIALER intent to show a dialog to allow the user to change the default Phone application, if the device implementation reports android.hardware.telephony .
- MUST honor the android.settings.ACTION_VOICE_INPUT_SETTINGS intent when the device supports the VoiceInteractionService and show a default app settings menu for voice input and assist.

3.3. Native API Compatibility

Native code compatibility is challenging. For this reason, device implementers are **STRONGLY RECOMMENDED** to use the implementations of the libraries listed below from the upstream Android Open Source Project.

3.3.1. Application Binary Interfaces

Managed Dalvik bytecode can call into native code provided in the application .apk file as an ELF .so file compiled for the appropriate device hardware architecture. As native code is highly dependent on the underlying processor technology, Android defines a number of Application Binary Interfaces (ABIs) in the Android NDK. Device implementations MUST be compatible with one or more defined ABIs, and MUST implement compatibility with the Android NDK, as below.

If a device implementation includes support for an Android ABI, it:

- MUST include support for code running in the managed environment to call into native code, using the standard Java Native Interface (JNI) semantics.
- MUST be source-compatible (i.e. header compatible) and binary-compatible (for the ABI) with each required library in the list below.
- MUST support the equivalent 32-bit ABI if any 64-bit ABI is supported.
- MUST accurately report the native Application Binary Interface (ABI) supported by the device, via the android.os.Build.SUPPORTED_ABIS, android.os.Build.SUPPORTED_32_BIT_ABIS, and android.os.Build.SUPPORTED_64_BIT_ABIS parameters, each a comma separated list of ABIs ordered from the most to the least preferred one.
- MUST report, via the above parameters, only those ABIs documented and described in the latest version of the Android NDK ABI Management documentation, and MUST include support for the Advanced SIMD (a.k.a. NEON) extension.
- SHOULD be built using the source code and header files available in the upstream Android Open Source Project

Note that future releases of the Android NDK may introduce support for additional ABIs. If a device implementation is not compatible with an existing predefined ABI, it MUST NOT report support for any ABIs at all.

The following native code APIs MUST be available to apps that include native code:

- libandroid.so (native Android activity support)
- libc (C library)
- libcamera2ndk.so
- libdl (dynamic linker)
- libEGL.so (native OpenGL surface management)
- libGLESv1_CM.so (OpenGL ES 1.x)
- libGLESv2.so (OpenGL ES 2.0)
- libGLESv3.so (OpenGL ES 3.x)
- libicui18n.so
- libicuuc.so
- libjnigraphics.so
- liblog (Android logging)
- libmediandk.so (native media APIs support)
- libm (math library)
- libOpenMAXAL.so (OpenMAX AL 1.0.1 support)
- libOpenSLES.so (OpenSL ES 1.0.1 audio support)
- libRS.so
- libstdc++ (Minimal support for C++)
- libvukan.so (Vulkan)
- libz (Zlib compression)
- JNI interface
- Support for OpenGL, as described below

For the native libraries listed above, the device implementation MUST NOT add or remove the public functions.

Native libraries not listed above but implemented and provided in AOSP as system libraries are reserved and MUST NOT be exposed to third-party apps targeting API level 24 or higher.

Device implementations MAY add non-AOSP libraries and expose them directly as an API to third-party apps but the additional libraries SHOULD be in `/vendor/lib` or `/vendor/lib64` and MUST be listed in `/vendor/etc/public.libraries.txt` .

Note that device implementations MUST include libGLESv3.so and in turn, MUST export all the OpenGL ES 3.1 and Android Extension Pack function symbols as defined in the NDK release android-24. Although all the symbols must be present, only the corresponding functions for OpenGL ES versions and extensions actually supported by the device must be fully implemented.

3.3.1.1. Graphic Libraries

Vulkan is a low-overhead, cross-platform API for high-performance 3D graphics. Device implementations, even if not including support of the Vulkan APIs, MUST satisfy the following requirements:

- It MUST always provide a native library named `libvulkan.so` which exports function symbols for the core Vulkan 1.0 API as well as the `VK_KHR_surface` , `VK_KHR_android_surface` , and `VK_KHR_swapchain` extensions.

Device implementations, if including support of the Vulkan APIs:

- MUST report, one or more `VkPhysicalDevices` through the `vkEnumeratePhysicalDevices` call.
- Each enumerated `VkPhysicalDevices` MUST fully implement the Vulkan 1.0 API.

- MUST report the correct PackageManager#FEATURE_VULKAN_HARDWARE_LEVEL and PackageManager#FEATURE_VULKAN_HARDWARE_VERSION feature flags.
- MUST enumerate layers, contained in native libraries named libVkLayer*.so in the application package's native library directory, through the vkEnumerateInstanceLayerProperties and vkEnumerateDeviceLayerProperties functions in libvulkan.so
- MUST NOT enumerate layers provided by libraries outside of the application package, or provide other ways of tracing or intercepting the Vulkan API, unless the application has the android:debuggable="true" attribute.

Device implementations, if not including support of the Vulkan APIs:

- MUST report 0 VkPhysicalDevices through the vkEnumeratePhysicalDevices call.
- MUST NOT delare any of the Vulkan feature flags PackageManager#FEATURE_VULKAN_HARDWARE_LEVEL and PackageManager#FEATURE_VULKAN_HARDWARE_VERSION .

3.3.2. 32-bit ARM Native Code Compatibility

The ARMv8 architecture deprecates several CPU operations, including some operations used in existing native code. On 64-bit ARM devices, the following deprecated operations MUST remain available to 32-bit native ARM code, either through native CPU support or through software emulation:

- SWP and SWPB instructions
- SETEND instruction
- CP15ISB, CP15DSB, and CP15DMB barrier operations

Legacy versions of the Android NDK used /proc/cpuinfo to discover CPU features from 32-bit ARM native code. For compatibility with applications built using this NDK, devices MUST include the following lines in /proc/cpuinfo when it is read by 32-bit ARM applications:

- "Features: ", followed by a list of any optional ARMv7 CPU features supported by the device.
- "CPU architecture: ", followed by an integer describing the device's highest supported ARM architecture (e.g., "8" for ARMv8 devices).

These requirements only apply when /proc/cpuinfo is read by 32-bit ARM applications. Devices SHOULD not alter /proc/cpuinfo when read by 64-bit ARM or non-ARM applications.

3.4. Web Compatibility

3.4.1. WebView Compatibility

Android Watch devices MAY, but all other device implementations MUST provide a complete implementation of the android.webkit.Webview API.

The platform feature android.software.webview MUST be reported on any device that provides a complete implementation of the android.webkit.WebView API, and MUST NOT be reported on devices without a complete implementation of the API. The Android Open Source implementation uses code from the Chromium Project to implement the android.webkit.WebView. Because it is not feasible to develop a comprehensive test suite for a web rendering system, device implementers MUST use the specific upstream build of Chromium in the WebView implementation. Specifically:

- Device android.webkit.WebView implementations MUST be based on the Chromium build

from the upstream Android Open Source Project for Android 7.1. This build includes a specific set of functionality and security fixes for the WebView.

- The user agent string reported by the WebView MUST be in this format:

  Mozilla/5.0 (Linux; Android $(VERSION); $(MODEL) Build/$(BUILD); wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 $(CHROMIUM_VER) Mobile Safari/537.36
  - The value of the $(VERSION) string MUST be the same as the value for android.os.Build.VERSION.RELEASE.
  - The value of the $(MODEL) string MUST be the same as the value for android.os.Build.MODEL.
  - The value of the $(BUILD) string MUST be the same as the value for android.os.Build.ID.
  - The value of the $(CHROMIUM_VER) string MUST be the version of Chromium in the upstream Android Open Source Project.
  - Device implementations MAY omit Mobile in the user agent string.

The WebView component SHOULD include support for as many HTML5 features as possible and if it supports the feature SHOULD conform to the HTML5 specification.

3.4.2. Browser Compatibility

Android Television, Watch, and Android Automotive implementations MAY omit a browser application, but MUST support the public intent patterns as described in section 3.2.3.1. All other types of device implementations MUST include a standalone Browser application for general user web browsing.

The standalone Browser MAY be based on a browser technology other than WebKit. However, even if an alternate Browser application is used, the android.webkit.WebView component provided to third-party applications MUST be based on WebKit, as described in section 3.4.1.

Implementations MAY ship a custom user agent string in the standalone Browser application.

The standalone Browser application (whether based on the upstream WebKit Browser application or a third-party replacement) SHOULD include support for as much of HTML5 as possible. Minimally, device implementations MUST support each of these APIs associated with HTML5:

- application cache/offline operation
- <video> tag
- geolocation

Additionally, device implementations MUST support the HTML5/W3C webstorage API and SHOULD support the HTML5/W3C IndexedDB API. Note that as the web development standards bodies are transitioning to favor IndexedDB over webstorage, IndexedDB is expected to become a required component in a future version of Android.

3.5. API Behavioral Compatibility

The behaviors of each of the API types (managed, soft, native, and web) must be consistent with the preferred implementation of the upstream Android Open Source Project. Some specific areas of compatibility are:

- Devices MUST NOT change the behavior or semantics of a standard intent.
- Devices MUST NOT alter the lifecycle or lifecycle semantics of a particular type of system component (such as Service, Activity, ContentProvider, etc.).
- Devices MUST NOT change the semantics of a standard permission.

The above list is not comprehensive. The Compatibility Test Suite (CTS) tests significant portions of the platform for behavioral compatibility, but not all. It is the responsibility of the implementer to ensure behavioral compatibility with the Android Open Source Project. For this reason, device implementers SHOULD use the source code available via the Android Open Source Project where possible, rather than re-implement significant parts of the system.

3.6. API Namespaces

Android follows the package and class namespace conventions defined by the Java programming language. To ensure compatibility with third-party applications, device implementers MUST NOT make any prohibited modifications (see below) to these package namespaces:

- java.*
- javax.*
- sun.*
- android.*
- com.android.*

Prohibited modifications include :

- Device implementations MUST NOT modify the publicly exposed APIs on the Android platform by changing any method or class signatures, or by removing classes or class fields.
- Device implementers MAY modify the underlying implementation of the APIs, but such modifications MUST NOT impact the stated behavior and Java-language signature of any publicly exposed APIs.
- Device implementers MUST NOT add any publicly exposed elements (such as classes or interfaces, or fields or methods to existing classes or interfaces) to the APIs above.

A “publicly exposed element” is any construct that is not decorated with the“@hide” marker as used in the upstream Android source code. In other words, device implementers MUST NOT expose new APIs or alter existing APIs in the namespaces noted above. Device implementers MAY make internal-only modifications, but those modifications MUST NOT be advertised or otherwise exposed to developers.

Device implementers MAY add custom APIs, but any such APIs MUST NOT be in a namespace owned by or referring to another organization. For instance, device implementers MUST NOT add APIs to the com.google.* or similar namespace: only Google may do so. Similarly, Google MUST NOT add APIs to other companies' namespaces. Additionally, if a device implementation includes custom APIs outside the standard Android namespace, those APIs MUST be packaged in an Android shared library so that only apps that explicitly use them (via the <uses-library> mechanism) are affected by the increased memory usage of such APIs.

If a device implementer proposes to improve one of the package namespaces above (such as by adding useful new functionality to an existing API, or adding a new API), the implementer SHOULD visit source.android.com and begin the process for contributing changes and code, according to the information on that site.

Note that the restrictions above correspond to standard conventions for naming APIs in the Java programming language; this section simply aims to reinforce those conventions and make them binding through inclusion in this Compatibility Definition.

3.7. Runtime Compatibility

Device implementations MUST support the full Dalvik Executable (DEX) format and Dalvik bytecode specification and semantics. Device implementers SHOULD use ART, the reference upstream

implementation of the Dalvik Executable Format, and the reference implementation's package management system.

Device implementations MUST configure Dalvik runtimes to allocate memory in accordance with the upstream Android platform, and as specified by the following table. (See section 7.1.1 for screen size and screen density definitions.) Note that memory values specified below are considered minimum values and device implementations MAY allocate more memory per application.

Screen Layout	Screen Density	Minimum Application Memory
Android Watch	120 dpi (ldpi)	32MB
	160 dpi (mdpi)	
	213 dpi (tvdpi)	
	240 dpi (hdpi)	36MB
	280 dpi (280dpi)	
	320 dpi (xhdpi)	48MB
	360 dpi (360dpi)	
	400 dpi (400dpi)	56MB
	420 dpi (420dpi)	64MB
	480 dpi (xxhdpi)	88MB
	560 dpi (560dpi)	112MB
	640 dpi (xxxhdpi)	154MB
small/normal	120 dpi (ldpi)	32MB
	160 dpi (mdpi)	
	213 dpi (tvdpi)	48MB
	240 dpi (hdpi)	
	280 dpi (280dpi)	
	320 dpi (xhdpi)	80MB
	360 dpi (360dpi)	
	400 dpi (400dpi)	96MB
	420 dpi (420dpi)	112MB
	480 dpi (xxhdpi)	128MB
	560 dpi (560dpi)	192MB
	640 dpi (xxxhdpi)	256MB
large	120 dpi (ldpi)	32MB
	160 dpi (mdpi)	48MB
	213 dpi (tvdpi)	80MB
	240 dpi (hdpi)	
	280 dpi (280dpi)	96MB
	320 dpi (xhdpi)	128MB
	360 dpi (360dpi)	160MB

	400 dpi (400dpi) 420 dpi (420dpi)	192MB 228MB
	480 dpi (xxhdpi)	256MB
	560 dpi (560dpi)	384MB
	640 dpi (xxxhdpi)	512MB
xlarge	120 dpi (ldpi)	48MB
	160 dpi (mdpi)	80MB
	213 dpi (tvdpi)	96MB
	240 dpi (hdpi)	
	280 dpi (280dpi)	144MB
	320 dpi (xhdpi)	192MB
	360 dpi (360dpi)	240MB
	400 dpi (400dpi)	288MB
	420 dpi (420dpi)	336MB
	480 dpi (xxhdpi)	384MB
	560 dpi (560dpi)	576MB
	640 dpi (xxxhdpi)	768MB

3.8. User Interface Compatibility

3.8.1. Launcher (Home Screen)

Android includes a launcher application (home screen) and support for third-party applications to replace the device launcher (home screen). Device implementations that allow third-party applications to replace the device home screen MUST declare the platform feature android.software.home_screen.

3.8.2. Widgets

Widgets are optional for all Android device implementations, but SHOULD be supported on Android Handheld devices.

Android defines a component type and corresponding API and lifecycle that allows applications to expose an "AppWidget" to the end user, a feature that is STRONGLY RECOMMENDED to be supported on Handheld Device implementations. Device implementations that support embedding widgets on the home screen MUST meet the following requirements and declare support for platform feature android.software.app_widgets.

- Device launchers MUST include built-in support for AppWidgets and expose user interface affordances to add, configure, view, and remove AppWidgets directly within the Launcher.
- Device implementations MUST be capable of rendering widgets that are 4 x 4 in the standard grid size. See the App Widget Design Guidelines in the Android SDK documentation for details.
- Device implementations that include support for lock screen MAY support application widgets on the lock screen.
- SHOULD trigger the fast-switch action between the two most recently used apps, when the recents function key is tapped twice.
- SHOULD trigger the split-screen multiwindow-mode, if supported, when the recents

functions key is long pressed.

3.8.3. Notifications

Android includes APIs that allow developers to notify users of notable events using hardware and software features of the device.

Some APIs allow applications to perform notifications or attract attention using hardware—specifically sound, vibration, and light. Device implementations MUST support notifications that use hardware features, as described in the SDK documentation, and to the extent possible with the device implementation hardware. For instance, if a device implementation includes a vibrator, it MUST correctly implement the vibration APIs. If a device implementation lacks hardware, the corresponding APIs MUST be implemented as no-ops. This behavior is further detailed in section 7.

Additionally, the implementation MUST correctly render all resources (icons, animation files etc.) provided for in the APIs, or in the Status/System Bar icon style guide, which in the case of an Android Television device includes the possibility to not display the notifications. Device implementers MAY provide an alternative user experience for notifications than that provided by the reference Android Open Source implementation; however, such alternative notification systems MUST support existing notification resources, as above.

> Android Automotive implementations MAY manage the visibility and timing of notifications to mitigate driver distraction, but MUST display notifications that use CarExtender when requested by applications.

Android includes support for various notifications, such as:

- **Rich notifications** . Interactive Views for ongoing notifications.
- **Heads-up notifications** . Interactive Views users can act on or dismiss without leaving the current app.
- **Lock screen notifications** . Notifications shown over a lock screen with granular control on visibility.

Android device implementations, when such notifications are made visible, MUST properly execute Rich and Heads-up notifications and include the title/name, icon, text as documented in the Android APIs.

Android includes Notification Listener Service APIs that allow apps (once explicitly enabled by the user) to receive a copy of all notifications as they are posted or updated. Device implementations MUST correctly and promptly send notifications in their entirety to all such installed and user-enabled listener services, including any and all metadata attached to the Notification object.

Handheld device implementations MUST support the behaviors of updating, removing, replying to, and bundling notifications as described in this section.

Also, handheld device implementations MUST provide:

- The ability to control notifications directly in the notification shade.
- The visual affordance to trigger the control panel in the notification shade.
- The ability to BLOCK, MUTE and RESET notification preference from a package, both in the inline control panel as well as in the settings app.

All 6 direct subclasses of the Notification.Style class MUST be supported as described in the SDK documents.

Device implementations that support the DND (Do not Disturb) feature MUST meet the following requirements:

- MUST implement an activity where the user can grant or deny the app access to DND policy configurations in response to the intent

ACTION_NOTIFICATION_POLICY_ACCESS_SETTINGS.

- MUST display Automatic DND rules created by applications alongside the user-created and pre-defined rules.
- MUST honor the suppressedVisualEffects values passed along the NotificationManager.Policy .

3.8.4. Search

Android includes APIs that allow developers to incorporate search into their applications and expose their application's data into the global system search. Generally speaking, this functionality consists of a single, system-wide user interface that allows users to enter queries, displays suggestions as users type, and displays results. The Android APIs allow developers to reuse this interface to provide search within their own apps and allow developers to supply results to the common global search user interface.

Android device implementations SHOULD include global search, a single, shared, system-wide search user interface capable of real-time suggestions in response to user input. Device implementations SHOULD implement the APIs that allow developers to reuse this user interface to provide search within their own applications. Device implementations that implement the global search interface MUST implement the APIs that allow third-party applications to add suggestions to the search box when it is run in global search mode. If no third-party applications are installed that make use of this functionality, the default behavior SHOULD be to display web search engine results and suggestions.

Android device implementations SHOULD, and Android Automotive implementations MUST, implement an assistant on the device to handle the Assist action.

Android also includes the Assist APIs to allow applications to elect how much information of the current context is shared with the assistant on the device. Device implementations supporting the Assist action MUST indicate clearly to the end user when the context is shared by displaying a white light around the edges of the screen. To ensure clear visibility to the end user, the indication MUST meet or exceed the duration and brightness of the Android Open Source Project implementation.

This indication MAY be disabled by default for preinstalled apps using the Assist and VoiceInteractionService API, if all following requirements are met:

- The preinstalled app MUST request the context to be shared only when the user invoked the app by one of the following means, and the app is running in the foreground:
  - hotword invocation
  - input of the ASSIST navigation key/button/gesture
- The device implementation MUST provide an affordance to enable the indication, less than two navigations away from (the default voice input and assistant app settings menu) section 3.2.3.5.

3.8.5. Toasts

Applications can use the "Toast" API to display short non-modal strings to the end user that disappear after a brief period of time. Device implementations MUST display Toasts from applications to end users in some high-visibility manner.

3.8.6. Themes

Android provides "themes" as a mechanism for applications to apply styles across an entire Activity or application.

Android includes a "Holo" theme family as a set of defined styles for application developers to use if they want to match the Holo theme look and feel as defined by the Android SDK. Device

implementations MUST NOT alter any of the Holo theme attributes exposed to applications.

Android includes a “Material” theme family as a set of defined styles for application developers to use if they want to match the design theme’s look and feel across the wide variety of different Android device types. Device implementations MUST support the “Material” theme family and MUST NOT alter any of the Material theme attributes or their assets exposed to applications.

Android also includes a “Device Default” theme family as a set of defined styles for application developers to use if they want to match the look and feel of the device theme as defined by the device implementer. Device implementations MAY modify the Device Default theme attributes exposed to applications.

Android supports a variant theme with translucent system bars, which allows application developers to fill the area behind the status and navigation bar with their app content. To enable a consistent developer experience in this configuration, it is important the status bar icon style is maintained across different device implementations. Therefore, Android device implementations MUST use white for system status icons (such as signal strength and battery level) and notifications issued by the system, unless the icon is indicating a problematic status or an app requests a light status bar using the SYSTEM_UI_FLAG_LIGHT_STATUS_BAR flag. When an app requests a light status bar, Android device implementations MUST change the color of the system status icons to black (for details, refer to R.style).

3.8.7. Live Wallpapers

Android defines a component type and corresponding API and lifecycle that allows applications to expose one or more “Live Wallpapers” to the end user. Live wallpapers are animations, patterns, or similar images with limited input capabilities that display as a wallpaper, behind other applications.

Hardware is considered capable of reliably running live wallpapers if it can run all live wallpapers, with no limitations on functionality, at a reasonable frame rate with no adverse effects on other applications. If limitations in the hardware cause wallpapers and/or applications to crash, malfunction, consume excessive CPU or battery power, or run at unacceptably low frame rates, the hardware is considered incapable of running live wallpaper. As an example, some live wallpapers may use an OpenGL 2.0 or 3.x context to render their content. Live wallpaper will not run reliably on hardware that does not support multiple OpenGL contexts because the live wallpaper use of an OpenGL context may conflict with other applications that also use an OpenGL context.

Device implementations capable of running live wallpapers reliably as described above SHOULD implement live wallpapers, and when implemented MUST report the platform feature flag android.software.live_wallpaper.

3.8.8. Activity Switching

> As the Recent function navigation key is OPTIONAL, the requirement to implement the overview screen is OPTIONAL for Android Watch and Android Automotive implementations, and RECOMMENDED for Android Television devices. There SHOULD still be a method to switch between activities on Android Automotive implementations.

The upstream Android source code includes the overview screen, a system-level user interface for task switching and displaying recently accessed activities and tasks using a thumbnail image of the application’s graphical state at the moment the user last left the application. Device implementations including the recents function navigation key as detailed in section 7.2.3 MAY alter the interface but MUST meet the following requirements:

- MUST support at least up to 20 displayed activities.
- MUST at least display the title of 4 activities at a time.
- MUST implement the screen pinning behavior and provide the user with a settings menu to toggle the feature.
- SHOULD display highlight color, icon, screen title in recents.

- SHOULD display a closing affordance ("x") but MAY delay this until user interacts with screens.
- SHOULD implement a shortcut to switch easily to the previous activity
- MAY display affiliated recents as a group that moves together.

Device implementations are STRONGLY RECOMMENDED to use the upstream Android user interface (or a similar thumbnail-based interface) for the overview screen.

3.8.9. Input Management

Android includes support for Input Management and support for third-party input method editors. Device implementations that allow users to use third-party input methods on the device MUST declare the platform feature android.software.input_methods and support IME APIs as defined in the Android SDK documentation.

Device implementations that declare the android.software.input_methods feature MUST provide a user-accessible mechanism to add and configure third-party input methods. Device implementations MUST display the settings interface in response to the android.settings.INPUT_METHOD_SETTINGS intent.

3.8.10. Lock Screen Media Control

The Remote Control Client API is deprecated from Android 5.0 in favor of the Media Notification Template that allows media applications to integrate with playback controls that are displayed on the lock screen. Device implementations that support a lock screen, unless an Android Automotive or Watch implementation, MUST display the Lock screen Notifications including the Media Notification Template.

3.8.11. Screen savers (previously Dreams)

Android includes support for interactive screensavers, previously referred to as Dreams. Screen savers allow users to interact with applications when a device connected to a power source is idle or docked in a desk dock. Android Watch devices MAY implement screen savers, but other types of device implementations SHOULD include support for screen savers and provide a settings option for users toconfigure screen savers in response to the android.settings.DREAM_SETTINGS intent.

3.8.12. Location

When a device has a hardware sensor (e.g. GPS) that is capable of providing the location coordinates, location modes MUST be displayed in the Location menu within Settings.

3.8.13. Unicode and Font

Android includes support for the emoji characters defined in Unicode 9.0. All device implementations MUST be capable of rendering these emoji characters in color glyph and when Android device implementations include an IME, it SHOULD provide an input method to the user for these emoji characters.

Android handheld devices SHOULD support the skin tone and diverse family emojis as specified in the Unicode Technical Report #51.

Android includes support for Roboto 2 font with different weights—sans-serif-thin, sans-serif-light, sans-serif-medium, sans-serif-black, sans-serif-condensed, sans-serif-condensed-light—which MUST all be included for the languages available on the device and full Unicode 7.0 coverage of Latin, Greek, and Cyrillic, including the Latin Extended A, B, C, and D ranges, and all glyphs in the currency symbols block of Unicode 7.0.

3.8.14. Multi-windows

A device implementation MAY choose not to implement any multi-window modes, but if it has the capability to display multiple activities at the same time it MUST implement such multi-window mode(s) in accordance with the application behaviors and APIs described in the Android SDK multi-window mode support documentation and meet the following requirements:

- Applications can indicate whether they are capable of operating in multi-window mode in the AndroidManifest.xml file, either explicitly via the android:resizeableActivity attribute or implicitly by having the targetSdkVersion > 24. Apps that explicitly set this attribute to false in their manifest MUST not be launched in multi-window mode. Apps that don't set the attribute in their manifest file (targetSdkVersion < 24) can be launched in multi-window mode, but the system MUST provide warning that the app may not work as expected in multi-window mode.
- Device implementations MUST NOT offer split-screen or freeform mode if both the screen height and width is less than 440 dp.
- Device implementations with screen size xlarge SHOULD support freeform mode.
- Android Television device implementations MUST support picture-in-picture (PIP) mode multi-window and place the PIP multi-window in the top right corner when PIP is ON.
- Device implementations with PIP mode multi-window support MUST allocate at least 240x135 dp for the PIP window.
- If the PIP multi-window mode is supported the KeyEvent.KEYCODE_WINDO key MUST be used to control the PIP window; otherwise, the key MUST be available to the foreground activity.

3.9. Device Administration

Android includes features that allow security-aware applications to perform device administration functions at the system level, such as enforcing password policies or performing remote wipe, through the Android Device Administration API]. Device implementations MUST provide an implementation of the DevicePolicyManager class. Device implementations that supports a secure lock screen MUST implement the full range of device administration policies defined in the Android SDK documentation and report the platform feature android.software.device_admin.

3.9.1 Device Provisioning

3.9.1.1 Device owner provisioning

If a device implementation declares the android.software.device_admin feature then it MUST implement the provisioning of the Device Owner app of a Device Policy Client (DPC) application as indicated below:

- When the device implementation has no user data configured yet, it:
  - MUST report true for DevicePolicyManager.isProvisioningAllowed(ACTION_PROVISION_MANAGED_DEVICE) .
  - MUST enroll the DPC application as the Device Owner app in response to the intent action android.app.action.PROVISION_MANAGED_DEVICE .
  - MUST enroll the DPC application as the Device Owner app if the device declares Near-Field Communications (NFC) support via the feature flag android.hardware.nfc and receives an NFC message containing a record with MIME type MIME_TYPE_PROVISIONING_NFC .
- When the device implementation has user data, it:

- MUST report false for the DevicePolicyManager.isProvisioningAllowed(ACTION_PROVISION_MANAGED_DEVICE) .
- MUST not enroll any DPC application as the Device Owner App any more.

Device implementations MAY have a preinstalled application performing device administration functions but this application MUST NOT be set as the Device Owner app without explicit consent or action from the user or the administrator of the device.

3.9.1.2 Managed profile provisioning

If a device implementation declares the android.software.managed_users, it MUST be possible to enroll a Device Policy Controller (DPC) application as the owner of a new Managed Profile.

The managed profile provisioning process (the flow initiated by android.app.action.PROVISION_MANAGED_PROFILE) user experience MUST align with the AOSP implementation.

Device implementations MUST provide the following user affordances within the Settings user interface to indicate to the user when a particular system function has been disabled by the Device Policy Controller (DPC):

- A consistent icon or other user affordance (for example the upstream AOSP info icon) to represent when a particular setting is restricted by a Device Admin.
- A short explanation message, as provided by the Device Admin via the [setShortSupportMessage] (https://developer.android.com/reference/android/app/admin/DevicePolicyManager.html#setShortSuppo java.lang.CharSequence).
- The DPC application's icon.

3.9.2 Managed Profile Support

Managed profile capable devices are those devices that:

- Declare android.software.device_admin (see section 3.9 Device Administration).
- Are not low RAM devices (see section 7.6.1).
- Allocate internal (non-removable) storage as shared storage (see section 7.6.2).

Managed profile capable devices MUST:

- Declare the platform feature flag android.software.managed_users .
- Support managed profiles via the android.app.admin.DevicePolicyManager APIs.
- Allow one and only one managed profile to be created.
- Use an icon badge (similar to the AOSP upstream work badge) to represent the managed applications and widgets and other badged UI elements like Recents & Notifications.
- Display a notification icon (similar to the AOSP upstream work badge) to indicate when user is within a managed profile application.
- Display a toast indicating that the user is in the managed profile if and when the device wakes up (ACTION_USER_PRESENT) and the foreground application is within the managed profile.
- Where a managed profile exists, show a visual affordance in the Intent 'Chooser' to allow the user to forward the intent from the managed profile to the primary user or vice versa, if enabled by the Device Policy Controller.
- Where a managed profile exists, expose the following user affordances for both the

primary user and the managed profile:
    - Separate accounting for battery, location, mobile data and storage usage for the primary user and managed profile.
    - Independent management of VPN Applications installed within the primary user or managed profile.
    - Independent management of applications installed within the primary user or managed profile.
    - Independent management of accounts within the primary user or managed profile.
- Ensure the preinstalled dialer, contacts and messaging applications can search for and look up caller information from the managed profile (if one exists) alongside those from the primary profile, if the Device Policy Controller permits it. When contacts from the managed profile are displayed in the preinstalled call log, in-call UI, in-progress and missed-call notifications, contacts and messaging apps they SHOULD be badged with the same badge used to indicate managed profile applications.
- MUST ensure that it satisfies all the security requirements applicable for a device with multiple users enabled (see section 9.5), even though the managed profile is not counted as another user in addition to the primary user.
- Support the ability to specify a separate lock screen meeting the following requirements to grant access to apps running in a managed profile.
    - Device implementations MUST honor the DevicePolicyManager.ACTION_SET_NEW_PASSWORD intent and show an interface to configure a separate lock screen credential for the managed profile.
    - The lock screen credentials of the managed profile MUST use the same credential storage and management mechanisms as the parent profile, as documented on the Android Open Source Project Site
    - The DPC password policies MUST apply to only the managed profile's lock screen credentials unless called upon the DevicePolicyManager instance returned by getParentProfileInstance..

3.10. Accessibility

Android provides an accessibility layer that helps users with disabilities to navigate their devices more easily. In addition, Android provides platform APIs that enable accessibility service implementations to receive callbacks for user and system events and generate alternate feedback mechanisms, such as text-to-speech, haptic feedback, and trackball/d-pad navigation.

Device implementations include the following requirements:

- Android Automotive implementations SHOULD provide an implementation of the Android accessibility framework consistent with the default Android implementation.
- Device implementations (Android Automotive excluded) MUST provide an implementation of the Android accessibility framework consistent with the default Android implementation.
- Device implementations (Android Automotive excluded) MUST support third-party accessibility service implementations through the android.accessibilityservice APIs.
- Device implementations (Android Automotive excluded) MUST generate AccessibilityEvents and deliver these events to all registered AccessibilityService implementations in a manner consistent with the default Android implementation
- Device implementations (Android Automotive and Android Watch devices with no audio output excluded), MUST provide a user-accessible mechanism to enable and disable accessibility services, and MUST display this interface in response to the android.provider.Settings.ACTION_ACCESSIBILITY_SETTINGS intent.
- Android device implementations with audio output are STRONGLY RECOMMENDED to

provide implementations of accessibility services on the device comparable in or exceeding functionality of the TalkBack** and Switch Access accessibility services (https://github.com/google/talkback).
- Android Watch devices with audio output SHOULD provide implementations of an accessibility service on the device comparable in or exceeding functionality of the TalkBack accessibility service (https://github.com/google/talkback).
- Device implementations SHOULD provide a mechanism in the out-of-box setup flow for users to enable relevant accessibility services, as well as options to adjust the font size, display size and magnification gestures.

** For languages supported by Text-to-speech.

Also, note that if there is a preloaded accessibility service, it MUST be a Direct Boot aware {directBootAware} app if the device has encrypted storage using File Based Encryption (FBE).

3.11. Text-to-Speech

Android includes APIs that allow applications to make use of text-to-speech (TTS) services and allows service providers to provide implementations of TTS services. Device implementations reporting the feature android.hardware.audio.output MUST meet these requirements related to the Android TTS framework.

Android Automotive implementations:

- MUST support the Android TTS framework APIs.
- MAY support installation of third-party TTS engines. If supported, partners MUST provide a user-accessible interface that allows the user to select a TTS engine for use at system level.

All other device implementations:

- MUST support the Android TTS framework APIs and SHOULD include a TTS engine supporting the languages available on the device. Note that the upstream Android open source software includes a full-featured TTS engine implementation.
- MUST support installation of third-party TTS engines.
- MUST provide a user-accessible interface that allows users to select a TTS engine for use at the system level.

3.12. TV Input Framework

The Android Television Input Framework (TIF) simplifies the delivery of live content to Android Television devices. TIF provides a standard API to create input modules that control Android Television devices. Android Television device implementations MUST support TV Input Framework.

Device implementations that support TIF MUST declare the platform feature android.software.live_tv.

3.12.1. TV App

Any device implementation that declares support for Live TV MUST have an installed TV application (TV App). The Android Open Source Project provides an implementation of the TV App.

The TV App MUST provide facilities to install and use TV Channels and meet the following requirements:

- Device implementations MUST allow third-party TIF-based inputs (third-party inputs) to be installed and managed.

- Device implementations MAY provide visual separation between pre-installed TIF-based inputs (installed inputs) and third-party inputs.
- Device implementations MUST NOT display the third-party inputs more than a single navigation action away from the TV App (i.e. expanding a list of third-party inputs from the TV App).

3.12.1.1. Electronic Program Guide

Android Television device implementations MUST show an informational and interactive overlay, which MUST include an electronic program guide (EPG) generated from the values in the TvContract.Programs fields. The EPG MUST meet the following requirements:

- The EPG MUST display information from all installed inputs and third-party inputs.
- The EPG MAY provide visual separation between the installed inputs and third-party inputs.
- The EPG is STRONGLY RECOMMENDED to display installed inputs and third-party inputs with equal prominence. The EPG MUST NOT display the third-party inputs more than a single navigation action away from the installed inputs on the EPG.
- On channel change, device implementations MUST display EPG data for the currently playing program.

3.12.1.2. Navigation

The TV App MUST allow navigation for the following functions via the D-pad, Back, and Home keys on the Android Television device's input device(s) (i.e. remote control, remote control application, or game controller):

- Changing TV channels
- Opening EPG
- Configuring and tuning to third-party TIF-based inputs
- Opening Settings menu

The TV App SHOULD pass key events to HDMI inputs through CEC.

3.12.1.3. TV input app linking

Android Television device implementations MUST support TV input app linking, which allows all inputs to provide activity links from the current activity to another activity (i.e. a link from live programming to related content). The TV App MUST show TV input app linking when it is provided.

3.12.1.4. Time shifting

Android Television device implementations MUST support time shifting, which allows the user to pause and resume live content. Device implementations MUST provide the user a way to pause and resume the currently playing program, if time shifting for that program is available.

3.12.1.5. TV recording

Android Television device implementations are STRONGLY RECOMMENDED to support TV recording. If the TV input supports recording, the EPG MAY provide a way to record a program if the recording of such a program is not prohibited. Device implementations SHOULD provide a user interface to play recorded programs.

3.13. Quick Settings

Android device implementations SHOULD include a Quick Settings UI component that allow quick access to frequently used or urgently needed actions.

Android includes the quicksettings API allowing third party apps to implement tiles that can be added by the user alongside the system-provided tiles in the Quick Settings UI component. If a device implementation has a Quick Settings UI component, it:

- MUST allow the user to add or remove tiles from a third-party app to Quick Settings.
- MUST NOT automatically add a tile from a third-party app directly to Quick Settings.
- MUST display all the user-added tiles from third-party apps alongside the system-provided quick setting tiles.

3.14. Vehicle UI APIs

3.14.1. Vehicle Media UI

Any device implementation that declares automotive support MUST include a UI framework to support third-party apps consuming the MediaBrowser and MediaSession APIs.

The UI framework supporting third-party apps that depend on MediaBrowser and MediaSession has the following visual requirements:

- MUST display MediaItem icons and notification icons unaltered.
- MUST display those items as described by MediaSession, e.g., metadata, icons, imagery.
- MUST show app title.
- MUST have drawer to present MediaBrowser hierarchy.

4. Application Packaging Compatibility

Device implementations MUST install and run Android ".apk" files as generated by the "aapt" tool included in the official Android SDK. For this reason device implementations SHOULD use the reference implementation's package management system.

The package manager MUST support verifying ".apk" files using the APK SignatureScheme v2.

Devices implementations MUST NOT extend either the .apk, Android Manifest, Dalvik bytecode, or RenderScript bytecode formats in such a way that would prevent those files from installing and running correctly on other compatible devices.

Device implementations MUST NOT allow apps other than the current "installer of record" for the package to silently uninstall the app without any prompt, as documented in the SDK for the DELETE_PACKAGE permission. The only exceptions are the system package verifier app handling PACKAGE_NEEDS_VERIFICATION intent and the storage manager app handling ACTION_MANAGE_STORAGE intent.

5. Multimedia Compatibility

5.1. Media Codecs

Device implementations—

- MUST support the core mediaformats specified in the Android SDK documentation, except where explicitly permitted in this document.

- MUST support the media formats, encoders, decoders, file types, and container formats defined in the tables below and reported via MediaCodecList.
- MUST also be able to decode all profiles reported in its CamcorderProfile
- MUST be able to decode all formats it can encode. This includes all bitstreams that its encoders generate.

Codecs SHOULD aim for minimum codec latency, in other words, codecs—

- SHOULD NOT consume and store input buffers and return input buffers only once processed
- SHOULD NOT hold onto decoded buffers for longer than as specified by the standard (e.g. SPS).
- SHOULD NOT hold onto encoded buffers longer than required by the GOP structure.

All of the codecs listed in the table below are provided as software implementations in the preferred Android implementation from the Android Open Source Project.

Please note that neither Google nor the Open Handset Alliance make any representation that these codecs are free from third-party patents. Those intending to use this source code in hardware or software products are advised that implementations of this code, including in open source software or shareware, may require patent licenses from the relevant patent holders.

5.1.1. Audio Codecs

Format/Codec	Encoder	Decoder	Details	Supported File Types/Container Formats
MPEG-4 AAC Profile (AAC LC)	REQUIRED [1]	REQUIRED	Support for mono/stereo/5.0/5.1 [2] content with standard sampling rates from 8 to 48 kHz.	• 3GPP (.3gp) • MPEG-4 (.mp4, .m4a) • ADTS raw AAC (.aac, decode in Android 3.1+, encode in Android 4.0+, ADIF not supported) • MPEG-TS (.ts, not seekable, Android 3.0+)
MPEG-4 HE AAC Profile (AAC+)	REQUIRED [1] (Android 4.1+)	REQUIRED	Support for mono/stereo/5.0/5.1 [2] content with standard sampling rates from 16 to 48 kHz.	
MPEG-4 HE AACv2 Profile (enhanced AAC+)		REQUIRED	Support for mono/stereo/5.0/5.1 [2] content with standard sampling rates from 16 to 48 kHz.	

AAC ELD (enhanced low delay AAC)	REQUIRED [1] (Android 4.1+)	REQUIRED (Android 4.1+)	Support for mono/stereo content with standard sampling rates from 16 to 48 kHz.	
AMR-NB	REQUIRED [3]	REQUIRED [3]	4.75 to 12.2 kbps sampled @ 8 kHz	3GPP (.3gp)
AMR-WB	REQUIRED [3]	REQUIRED [3]	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16 kHz	
FLAC		REQUIRED (Android 3.1+)	Mono/Stereo (no multichannel). Sample rates up to 48 kHz (but up to 44.1 kHz is RECOMMENDED on devices with 44.1 kHz output, as the 48 to 44.1 kHz downsampler does not include a low-pass filter). 16-bit RECOMMENDED; no dither applied for 24-bit.	FLAC (.flac) only
MP3		REQUIRED	Mono/Stereo 8-320Kbps constant (CBR) or variable bitrate (VBR)	MP3 (.mp3)
MIDI		REQUIRED	MIDI Type 0 and 1. DLS Version 1 and 2. XMF and Mobile XMF. Support for ringtone formats RTTTL/RTX, OTA, and iMelody	• Type 0 and 1 (.mid, .xmf, .mxmf) • RTTTL/RTX (.rtttl, .rtx) • OTA (.ota) • iMelody (.imy)
Vorbis		REQUIRED		• Ogg (.ogg) • Matroska (.mkv, Android 4.0+)
PCM/WAVE	REQUIRED [4] (Android 4.1+)	REQUIRED	16-bit linear PCM (rates up to limit of hardware). Devices MUST support sampling rates for raw PCM recording at 8000, 11025, 16000, and 44100 Hz frequencies.	WAVE (.wav)
Opus		REQUIRED (Android 5.0+)		Matroska (.mkv), Ogg(.ogg)

1 Required for device implementations that define android.hardware.microphone but optional for Android Watch device implementations.

2 Recording or playback MAY be performed in mono or stereo, but the decoding of AAC input buffers of multichannel streams (i.e. more than two channels) to PCM through the default AAC audio decoder in the android.media.MediaCodec API, the following MUST be supported:

- decoding is performed without downmixing (e.g. a 5.0 AAC stream must be decoded to five channels of PCM, a 5.1 AAC stream must be decoded to six channels of PCM),
- dynamic range metadata, as defined in "Dynamic Range Control (DRC)" in ISO/IEC

14496-3, and the android.media.MediaFormat DRC keys to configure the dynamic range-related behaviors of the audio decoder. The AAC DRC keys were introduced in API 21,and are: KEY_AAC_DRC_ATTENUATION_FACTOR, KEY_AAC_DRC_BOOST_FACTOR, KEY_AAC_DRC_HEAVY_COMPRESSION, KEY_AAC_DRC_TARGET_REFERENCE_LEVEL and KEY_AAC_ENCODED_TARGET_LEVEL

3 Required for Android Handheld device implementations.

4 Required for device implementations that define android.hardware.microphone, including Android Watch device implementations.

5.1.2. Image Codecs

Format/Codec	Encoder	Decoder	Details	Supported File Types/Container Formats
JPEG	REQUIRED	REQUIRED	Base+progressive	JPEG (.jpg)
GIF		REQUIRED		GIF (.gif)
PNG	REQUIRED	REQUIRED		PNG (.png)
BMP		REQUIRED		BMP (.bmp)
WebP	REQUIRED	REQUIRED		WebP (.webp)
Raw		REQUIRED		ARW (.arw), CR2 (.cr2), DNG (.dng), NEF (.nef), NRW (.nrw), ORF (.orf), PEF (.pef), RAF (.raf), RW2 (.rw2), SRW (.srw)

5.1.3. Video Codecs

- Codecs advertising HDR profile support MUST support HDR static metadata parsing and handling.
- If a media codec advertises intra refresh support, then it MUST support the refresh periods in the range of 10 - 60 frames and accurately operate within 20% of configured refresh period.
- Video codecs MUST support output and input bytebuffer sizes that accommodate the largest feasible compressed and uncompressed frame as dictated by the standard and configuration but also not overallocate.
- Video encoders and decoders MUST support YUV420 flexible color format (COLOR_FormatYUV420Flexible).

Format/Codec	Encoder	Decoder	Details	Supported File Types/ Container Formats
H.263	MAY	MAY		• 3GPP (.3gp) • MPEG-4 (.mp4)
H.264 AVC	REQUIRED [2]	REQUIRED [2]	See section 5.2 and 5.3 for details	• 3GPP (.3gp) • MPEG-4 (.mp4) • MPEG-2 TS (.ts, AAC audio only, not seekable, Android

				3.0+)
H.265 HEVC		REQUIRED [5]	See section 5.3 for details	MPEG-4 (.mp4)
MPEG-2		STRONGLY RECOMMENDED [6]	Main Profile	MPEG2-TS
MPEG-4 SP		REQUIRED [2]		3GPP (.3gp)
VP8 [3]	REQUIRED [2] (Android 4.3+)	REQUIRED [2] (Android 2.3.3+)	See section 5.2 and 5.3 for details	• WebM(.webm) • Matroska (.mkv, Android 4.0+) [4]
VP9		REQUIRED [2] (Android 4.4+)	See section 5.3 for details	• WebM(.webm) • Matroska (.mkv, Android 4.0+) [4]

1 Required for device implementations that include camera hardware and define android.hardware.camera or android.hardware.camera.front.

2 Required for device implementations except Android Watch devices.

3 For acceptable quality of web video streaming and video-conference services, device implementations SHOULD use a hardware VP8 codec that meets the requirements.

4 Device implementations SHOULD support writing Matroska WebM files.

5 STRONGLY RECOMMENDED for Android Automotive, optional for Android Watch, and required for all other device types.

6 Applies only to Android Television device implementations.

5.2. Video Encoding

Video codecs are optional for Android Watch device implementations.

H.264, VP8, VP9 and HEVC video encoders—

- MUST support dynamically configurable bitrates.
- SHOULD support variable frame rates, where video encoder SHOULD determine instantaneous frame duration based on the timestamps of input buffers, and allocate its bit bucket based on that frame duration.

H.263 and MPEG-4 video encoder SHOULD support dynamically configurable bitrates.

All video encoders SHOULD meet the following bitrate targets over two sliding windows:

- It SHOULD be not more than ~15% over the bitrate between intraframe (I-frame) intervals.
- It SHOULD be not more than ~100% over the bitrate over a sliding window of 1 second.

5.2.1. H.263

Android device implementations with H.263 encoders MUST support Baseline Profile Level 45.

5.2.2. H-264

Android device implementations with H.264 codec support&emdash;

- MUST support Baseline Profile Level 3.
  However, support for ASO (Arbitrary Slice Ordering), FMO (Flexible Macroblock Ordering) and RS (Redundant Slices) is OPTIONAL. Moreover, to maintain compatibility with other Android devices, it is RECOMMENDED that ASO, FMO and RS are not used for Baseline Profile by encoders.
- MUST support the SD (Standard Definition) video encoding profiles in the following table.
- SHOULD support Main Profile Level 4.
- SHOULD support the HD (High Definition) video encoding profiles as indicated in the following table.
- In addition, Android Television devices are STRONGLY RECOMMENDED to encode HD 1080p video at 30 fps.

	SD (Low quality)	SD (High quality)	HD 720p [1]	HD 1080p [1]
Video resolution	320 x 240 px	720 x 480 px	1280 x 720 px	1920 x 1080 px
Video frame rate	20 fps	30 fps	30 fps	30 fps
Video bitrate	384 Kbps	2 Mbps	4 Mbps	10 Mbps

1 When supported by hardware, but STRONGLY RECOMMENDED for Android Television devices.

5.2.3. VP8

Android device implementations with VP8 codec support MUST support the SD video encoding profiles and SHOULD support the following HD (High Definition) video encoding profiles.

	SD (Low quality)	SD (High quality)	HD 720p [1]	HD 1080p [1]
Video resolution	320 x 180 px	640 x 360 px	1280 x 720 px	1920 x 1080 px
Video frame rate	30 fps	30 fps	30 fps	30 fps
Video bitrate	800 Kbps	2 Mbps	4 Mbps	10 Mbps

1 When supported by hardware.

5.3. Video Decoding

Video codecs are optional for Android Watch device implementations.

Device implementations—

- MUST support dynamic video resolution and frame rate switching through the standard Android APIs within the same stream for all VP8, VP9, H.264, and H.265 codecs in real time and up to the maximum resolution supported by each codec on the device.
- Implementations that support the Dolby Vision decoder—
- MUST provide a Dolby Vision-capable extractor.
- MUST properly display Dolby Vision content on the device screen or on a standard video output port (e.g., HDMI).
- Implementations that provide a Dolby Vision-capable extractor MUST set the track index of backward-compatible base-layer(s) (if present) to be the same as the combined Dolby Vision layer's track index.

5.3.1. MPEG-2

Android device implementations with MPEG-2 decoders must support the Main Profile High Level.

5.3.2. H.263

Android device implementations with H.263 decoders MUST support Baseline Profile Level 30 and Level 45.

5.3.3. MPEG-4

Android device implementations with MPEG-4 decoders MUST support Simple Profile Level 3.

5.3.4. H.264

Android device implementations with H.264 decoders:

- MUST support Main Profile Level 3.1 and Baseline Profile.
  Support for ASO (Arbitrary Slice Ordering), FMO (Flexible Macroblock Ordering) and RS (Redundant Slices) is OPTIONAL.
- MUST be capable of decoding videos with the SD (Standard Definition) profiles listed in the following table and encoded with the Baseline Profile and Main Profile Level 3.1 (including 720p30).
- SHOULD be capable of decoding videos with the HD (High Definition) profiles as indicated in the following table.
- In addition, Android Television devices—
  - MUST support High Profile Level 4.2 and the HD 1080p60 decoding profile.
  - MUST be capable of decoding videos with both HD profiles as indicated in the following table and encoded with either the Baseline Profile, Main Profile, or the High Profile Level 4.2

	SD (Low quality)	SD (High quality)	HD 720p [1]	HD 1080p [1]
Video resolution	320 x 240 px	720 x 480 px	1280 x 720 px	1920 x 1080 px
Video frame rate	30 fps	30 fps	60 fps	30 fps (60 fps [2])
Video bitrate	800 Kbps	2 Mbps	8 Mbps	20 Mbps

1 REQUIRED for when the height as reported by the Display.getSupportedModes() method is equal or greater than the video resolution.

2 REQUIRED for Android Television device implementations.

5.3.5. H.265 (HEVC)

Android device implementations, when supporting H.265 codec as described in section 5.1.3:

- MUST support the Main Profile Level 3 Main tier and the SD video decoding profiles as indicated in the following table.
- SHOULD support the HD decoding profiles as indicated in the following table.
- MUST support the HD decoding profiles as indicated in the following table if there is a hardware decoder.
- In addition, Android Television devices:
- MUST support the HD 720p decoding profile.

- STRONGLY RECOMMENDED to support the HD 1080p decoding profile. If the HD 1080p decoding profile is supported, it MUST support the Main Profile Level 4.1 Main tier.
- SHOULD support the UHD decoding profile. If the UHD decoding profile is supported the codec MUST support Main10 Level 5 Main Tier profile.

	SD (Low quality)	SD (High quality)	HD 720p	HD 1080p	UHD
Video resolution	352 x 288 px	720 x 480 px	1280 x 720 px	1920 x 1080 px	3840 x 2160 px
Video frame rate	30 fps	30 fps	30 fps	30 fps (60 fps [1])	60 fps
Video bitrate	600 Kbps	1.6 Mbps	4 Mbps	10 Mbps	20 Mbps

1 REQUIRED for Android Television device implementations with H.265 hardware decoding.

5.3.6. VP8

Android device implementations, when supporting VP8 codec as described in section 5.1.3:

- MUST support the SD decoding profiles in the following table.
- SHOULD support the HD decoding profiles in the following table.
- Android Television devices MUST support the HD 1080p60 decoding profile.

	SD (Low quality)	SD (High quality)	HD 720p [1]	HD 1080p [1]
Video resolution	320 x 180 px	640 x 360 px	1280 x 720 px	1920 x 1080 px
Video frame rate	30 fps	30 fps	30 fps (60 fps [2])	30 (60 fps [2])
Video bitrate	800 Kbps	2 Mbps	8 Mbps	20 Mbps

1 REQUIRED for when the height as reported by the Display.getSupportedModes() method is equal or greater than the video resolution.

2 REQUIRED for Android Television device implementations.

5.3.7. VP9

Android device implementations, when supporting VP9 codec as described in section 5.1.3:

- MUST support the SD video decoding profiles as indicated in the following table.
- SHOULD support the HD decoding profiles as indicated in the following table.
- MUST support the HD decoding profiles as indicated in the following table, if there is a hardware decoder.
- In addition, Android Television devices:
  - MUST support the HD 720p decoding profile.
  - STRONGLY RECOMMENDED to support the HD 1080p decoding profile.
  - SHOULD support the UHD decoding profile. If the UHD video decoding profile is supported, it MUST support 8-bit color depth and SHOULD support VP9 Profile 2 (10-bit).

	SD (Low quality)	SD (High quality)	HD 720p	HD 1080p	UHD

Video resolution	320 x 180 px	640 x 360 px	1280 x 720 px	1920 x 1080 px	3840 x 2160 px
Video frame rate	30 fps	30 fps	30 fps	30 fps (60 fps [1])	60 fps
Video bitrate	600 Kbps	1.6 Mbps	4 Mbps	5 Mbps	20 Mbps

1 REQUIRED for Android Television device implementations with VP9 hardware decoding.

5.4. Audio Recording

While some of the requirements outlined in this section are stated as SHOULD since Android 4.3, the Compatibility Definition for a future version is planned to change these to MUST. Existing and new Android devices are **STRONGLY RECOMMENDED** to meet these requirements that are stated as SHOULD, or they will not be able to attain Android compatibility when upgraded to the future version.

5.4.1. Raw Audio Capture

Device implementations that declare android.hardware.microphone MUST allow capture of raw audio content with the following characteristics:

- **Format** : Linear PCM, 16-bit
- **Sampling rates** : 8000, 11025, 16000, 44100
- **Channels** : Mono

The capture for the above sample rates MUST be done without up-sampling, and any down-sampling MUST include an appropriate anti-aliasing filter.

Device implementations that declare android.hardware.microphone SHOULD allow capture of raw audio content with the following characteristics:

- **Format** : Linear PCM, 16-bit
- **Sampling rates** : 22050, 48000
- **Channels** : Stereo

If capture for the above sample rates is supported, then the capture MUST be done without up-sampling at any ratio higher than 16000:22050 or 44100:48000. Any up-sampling or down-sampling MUST include an appropriate anti-aliasing filter.

5.4.2. Capture for Voice Recognition

The android.media.MediaRecorder.AudioSource.VOICE_RECOGNITION audio source MUST support capture at one of the sampling rates, 44100 and 48000.

In addition to the above recording specifications, when an application has started recording an audio stream using the android.media.MediaRecorder.AudioSource.VOICE_RECOGNITION audio source:

- The device SHOULD exhibit approximately flat amplitude versus frequency characteristics: specifically, ±3 dB, from 100 Hz to 4000 Hz.
- Audio input sensitivity SHOULD be set such that a 90 dB sound power level (SPL) source at 1000 Hz yields RMS of 2500 for 16-bit samples.
- PCM amplitude levels SHOULD linearly track input SPL changes over at least a 30 dB range from -18 dB to +12 dB re 90 dB SPL at the microphone.
- Total harmonic distortion SHOULD be less than 1% for 1 kHz at 90 dB SPL input level at the microphone.

- Noise reduction processing, if present, MUST be disabled.
- Automatic gain control, if present, MUST be disabled.

If the platform supports noise suppression technologies tuned for speech recognition, the effect MUST be controllable from the android.media.audiofx.NoiseSuppressor API. Moreover, the UUID field for the noise suppressor's effect descriptor MUST uniquely identify each implementation of the noise suppression technology.

5.4.3. Capture for Rerouting of Playback

The android.media.MediaRecorder.AudioSource class includes the REMOTE_SUBMIX audio source. Devices that declare android.hardware.audio.output MUST properly implement the REMOTE_SUBMIX audio source so that when an application uses the android.media.AudioRecord API to record from this audio source, it can capture a mix of all audio streams except for the following:

- STREAM_RING
- STREAM_ALARM
- STREAM_NOTIFICATION

5.5. Audio Playback

Device implementations that declare android.hardware.audio.output MUST conform to the requirements in this section.

5.5.1. Raw Audio Playback

The device MUST allow playback of raw audio content with the following characteristics:

- **Format** : Linear PCM, 16-bit
- **Sampling rates** : 8000, 11025, 16000, 22050, 32000, 44100
- **Channels** : Mono, Stereo

The device SHOULD allow playback of raw audio content with the following characteristics:

- **Sampling rates** : 24000, 48000

5.5.2. Audio Effects

Android provides an API for audio effects for device implementations. Device implementations that declare the feature android.hardware.audio.output:

- MUST support the EFFECT_TYPE_EQUALIZER and EFFECT_TYPE_LOUDNESS_ENHANCER implementations controllable through the AudioEffect subclasses Equalizer, LoudnessEnhancer.
- MUST support the visualizer API implementation, controllable through the Visualizer class.
- SHOULD support the EFFECT_TYPE_BASS_BOOST, EFFECT_TYPE_ENV_REVERB, EFFECT_TYPE_PRESET_REVERB, and EFFECT_TYPE_VIRTUALIZER implementations controllable through the AudioEffect sub-classes BassBoost, EnvironmentalReverb, PresetReverb, and Virtualizer.

5.5.3. Audio Output Volume

Android Television device implementations MUST include support for system Master Volume and

digital audio output volume attenuation on supported outputs, except for compressed audio passthrough output (where no audio decoding is done on the device).

Android Automotive device implementations SHOULD allow adjusting audio volume separately per each audio stream using the content type or usage as defined by AudioAttributes and car audio usage as publicly defined in android.car.CarAudioManager .

5.6. Audio Latency

Audio latency is the time delay as an audio signal passes through a system. Many classes of applications rely on short latencies, to achieve real-time sound effects.

For the purposes of this section, use the following definitions:

- **output latency** . The interval between when an application writes a frame of PCM-coded data and when the corresponding sound is presented to environment at an on-device transducer or signal leaves the device via a port and can be observed externally.
- **cold output latency** . The output latency for the first frame, when the audio output system has been idle and powered down prior to the request.
- **continuous output latency** . The output latency for subsequent frames, after the device is playing audio.
- **input latency** . The interval between when a sound is presented by environment to device at an on-device transducer or signal enters the device via a port and when an application reads the corresponding frame of PCM-coded data.
- **lost input** . The initial portion of an input signal that is unusable or unavailable.
- **cold input latency** . The sum of lost input time and the input latency for the first frame, when the audio input system has been idle and powered down prior to the request.
- **continuous input latency** . The input latency for subsequent frames, while the device is capturing audio.
- **cold output jitter** . The variability among separate measurements of cold output latency values.
- **cold input jitter** . The variability among separate measurements of cold input latency values.
- **continuous round-trip latency** . The sum of continuous input latency plus continuous output latency plus one buffer period. The buffer period allows time for the app to process the signal and time for the app to mitigate phase difference between input and output streams.
- **OpenSL ES PCM buffer queue API** . The set of PCM-related OpenSL ES APIs within Android NDK.

Device implementations that declare android.hardware.audio.output are STRONGLY RECOMMENDED to meet or exceed these audio output requirements:

- cold output latency of 100 milliseconds or less
- continuous output latency of 45 milliseconds or less
- minimize the cold output jitter

If a device implementation meets the requirements of this section after any initial calibration when using the OpenSL ES PCM buffer queue API, for continuous output latency and cold output latency over at least one supported audio output device, it is STRONGLY RECOMMENDED to report support for low-latency audio, by reporting the feature android.hardware.audio.low_latency via the android.content.pm.PackageManager class. Conversely, if the device implementation does not meet these requirements it MUST NOT report support for low-latency audio.

Device implementations that include android.hardware.microphone are STRONGLY

RECOMMENDED to meet these input audio requirements:

- cold input latency of 100 milliseconds or less
- continuous input latency of 30 milliseconds or less
- continuous round-trip latency of 50 milliseconds or less
- minimize the cold input jitter

5.7. Network Protocols

Devices MUST support the media network protocols for audio and video playback as specified in the Android SDK documentation. Specifically, devices MUST support the following media network protocols:

- HTTP(S) progressive streaming
  All required codecs and container formats in section 5.1 MUST be supported over HTTP(S)
- HTTP Live Streaming draft protocol, Version 7
  The following media segment formats MUST be supported:

Segment formats	Reference(s)	Required codec support
MPEG-2 Transport Stream	ISO 13818	Video codecs: • H264 AVC • MPEG-4 SP • MPEG-2 See section 5.1.3 for details on H264 AVC, MPEG2-4 SP, and MPEG-2. Audio codecs: • AAC See section 5.1.1 for details on AAC and its variants.
AAC with ADTS framing and ID3 tags	ISO 13818-7	See section 5.1.1 for details on AAC and its variants
WebVTT	WebVTT	

- RTSP (RTP, SDP)
  The following RTP audio video profile and related codecs MUST be supported. For exceptions please see the table footnotes in section 5.1.

Profile name	Reference(s)	Required codec support
H264 AVC	RFC 6184	See section 5.1.3 for details on H264 AVC
MP4A-LATM	RFC 6416	See section 5.1.1 for details on AAC and its variants
H263-1998	RFC 3551 RFC 4629 RFC 2190	See section 5.1.3 for details on H263

~~H263-2000~~ AMR	~~RFC 4629~~ RFC 4867	~~See section 5.1.3 for details on H263~~ See section 5.1.1 for details on AMR-NB
AMR-WB	RFC 4867	See section 5.1.1 for details on AMR-WB
MP4V-ES	RFC 6416	See section 5.1.3 for details on MPEG-4 SP
mpeg4-generic	RFC 3640	See section 5.1.1 for details on AAC and its variants
MP2T	RFC 2250	See MPEG-2 Transport Stream underneath HTTP Live Streaming for details

5.8. Secure Media

Device implementations that support secure video output and are capable of supporting secure surfaces MUST declare support for Display.FLAG_SECURE. Device implementations that declare support for Display.FLAG_SECURE, if they support a wireless display protocol, MUST secure the link with a cryptographically strong mechanism such as HDCP 2.x or higher for Miracast wireless displays. Similarly if they support a wired external display, the device implementations MUST support HDCP 1.2 or higher. Android Television device implementations MUST support HDCP 2.2 for devices supporting 4K resolution and HDCP 1.4 or above for lower resolutions. The upstream Android open source implementation includes support for wireless (Miracast) and wired (HDMI) displays that satisfies this requirement.

5.9. Musical Instrument Digital Interface (MIDI)

If a device implementation supports the inter-app MIDI software transport (virtual MIDI devices), and it supports MIDI over *all* of the following MIDI-capable hardware transports for which it provides generic non-MIDI connectivity, it is STRONGLY RECOMMENDED to report support for feature android.software.midi via the android.content.pm.PackageManager class.

The MIDI-capable hardware transports are:

- USB host mode (section 7.7 USB)
- USB peripheral mode (section 7.7 USB)
- MIDI over Bluetooth LE acting in central role (section 7.4.3 Bluetooth)

Conversely, if the device implementation provides generic non-MIDI connectivity over a particular MIDI-capable hardware transport listed above, but does not support MIDI over that hardware transport, it MUST NOT report support for feature android.software.midi.

5.10. Professional Audio

If a device implementation meets *all* of the following requirements, it is STRONGLY RECOMMENDED to report support for feature android.hardware.audio.pro via the android.content.pm.PackageManager class.

- The device implementation MUST report support for feature android.hardware.audio.low_latency.
- The continuous round-trip audio latency, as defined in section 5.6 Audio Latency, MUST be 20 milliseconds or less and SHOULD be 10 milliseconds or less over at least one supported path.
- If the device includes a 4 conductor 3.5mm audio jack, the continuous round-trip audio latency MUST be 20 milliseconds or less over the audio jack path, and SHOULD be 10 milliseconds or less over at the audio jack path.

- The device implementation MUST include a USB port(s) supporting USB host mode and USB peripheral mode.
- The USB host mode MUST implement the USB audio class.
- If the device includes an HDMI port, the device implementation MUST support output in stereo and eight channels at 20-bit or 24-bit depth and 192 kHz without bit-depth loss or resampling.
- The device implementation MUST report support for feature android.software.midi.
- If the device includes a 4 conductor 3.5mm audio jack, the device implementation is STRONGLY RECOMMENDED to comply with section Mobile device (jack) specifications of the Wired Audio Headset Specification (v1.1).

Latencies and USB audio requirements MUST be met using the OpenSL ES PCM buffer queue API.

In addition, a device implementation that reports support for this feature SHOULD:

- Provide a sustainable level of CPU performance while audio is active.
- Minimize audio clock inaccuracy and drift relative to standard time.
- Minimize audio clock drift relative to the CPU CLOCK_MONOTONIC when both are active.
- Minimize audio latency over on-device transducers.
- Minimize audio latency over USB digital audio.
- Document audio latency measurements over all paths.
- Minimize jitter in audio buffer completion callback entry times, as this affects usable percentage of full CPU bandwidth by the callback.
- Provide zero audio underruns (output) or overruns (input) under normal use at reported latency.
- Provide zero inter-channel latency difference.
- Minimize MIDI mean latency over all transports.
- Minimize MIDI latency variability under load (jitter) over all transports.
- Provide accurate MIDI timestamps over all transports.
- Minimize audio signal noise over on-device transducers, including the period immediately after cold start.
- Provide zero audio clock difference between the input and output sides of corresponding end-points, when both are active. Examples of corresponding end-points include the on-device microphone and speaker, or the audio jack input and output.
- Handle audio buffer completion callbacks for the input and output sides of corresponding end-points on the same thread when both are active, and enter the output callback immediately after the return from the input callback. Or if it is not feasible to handle the callbacks on the same thread, then enter the output callback shortly after entering the input callback to permit the application to have a consistent timing of the input and output sides.
- Minimize the phase difference between HAL audio buffering for the input and output sides of corresponding end-points.
- Minimize touch latency.
- Minimize touch latency variability under load (jitter).

5.11. Capture for Unprocessed

Starting from Android 7.0, a new recording source has been added. It can be accessed using the android.media.MediaRecorder.AudioSource.UNPROCESSED audio source. In OpenSL ES, it can be accessed with the record preset SL_ANDROID_RECORDING_PRESET_UNPROCESSED .

A device MUST satisfy all of the following requirements to report support of the unprocessed audio source via the android.media.AudioManager property PROPERTY_SUPPORT_AUDIO_SOURCE_UNPROCESSED:

- The device MUST exhibit approximately flat amplitude-versus-frequency characteristics in the mid-frequency range: specifically ±10dB from 100 Hz to 7000 Hz.
- The device MUST exhibit amplitude levels in the low frequency range: specifically from ±20 dB from 5 Hz to 100 Hz compared to the mid-frequency range.
- The device MUST exhibit amplitude levels in the high frequency range: specifically from ±30 dB from 4000 Hz to 22 KHz compared to the mid-frequency range.
- Audio input sensitivity MUST be set such that a 1000 Hz sinusoidal tone source played at 94 dB Sound Pressure Level (SPL) yields a response with RMS of 520 for 16 bit-samples (or -36 dB Full Scale for floating point/double precision samples).
- SNR > 60 dB (difference between 94 dB SPL and equivalent SPL of self noise, A-weighted).
- Total harmonic distortion MUST be less than 1% for 1 kHZ at 90 dB SPL input level at the microphone.
- The only signal processing allowed in the path is a level multiplier to bring the level to desired range. This level multiplier MUST NOT introduce delay or latency to the signal path.
- No other signal processing is allowed in the path, such as Automatic Gain Control, High Pass Filter, or Echo Cancellation. If any signal processing is present in the architecture for any reason, it MUST be disabled and effectively introduce zero delay or extra latency to the signal path.

All SPL measurements are made directly next to the microphone under test.

For multiple microphone configurations, these requirements apply to each microphone.

It is STRONGLY RECOMMENDED that a device satisfy as many of the requirements for the signal path for the unprocessed recording source; however, a device must satisfy *all* of these requirements, listed above, if it claims to support the unprocessed audio source.

6. Developer Tools and Options Compatibility

6.1. Developer Tools

Device implementations MUST support the Android Developer Tools provided in the Android SDK. Android compatible devices MUST be compatible with:

- **Android Debug Bridge (adb)**
  - Device implementations MUST support all adb functions as documented in the Android SDK including dumpsys.
  - The device-side adb daemon MUST be inactive by default and there MUST be a user-accessible mechanism to turn on the Android Debug Bridge. If a device implementation omits USB peripheral mode, it MUST implement the Android Debug Bridge via local-area network (such as Ethernet or 802.11).
  - Android includes support for secure adb. Secure adb enables adb on known authenticated hosts. Device implementations MUST support secure adb.
- **Dalvik Debug Monitor Service (ddms)**
  - Device implementations MUST support all ddms features as documented in the Android SDK.
  - As ddms uses adb, support for ddms SHOULD be inactive by default, but MUST be supported whenever the user has activated the Android Debug Bridge, as above.
- **Monkey** . Device implementations MUST include the Monkey framework, and make it

available for applications to use.

- **SysTrace**
  - Device implementations MUST support systrace tool as documented in the Android SDK. Systrace must be inactive by default, and there MUST be a user-accessible mechanism to turn on Systrace.
  - Most Linux-based systems and Apple Macintosh systems recognize Android devices using the standard Android SDK tools, without additional support; however Microsoft Windows systems typically require a driver for new Android devices. (For instance, new vendor IDs and sometimes new device IDs require custom USB drivers for Windows systems.)
  - If a device implementation is unrecognized by the adb tool as provided in the standard Android SDK, device implementers MUST provide Windows drivers allowing developers to connect to the device using the adb protocol. These drivers MUST be provided for Windows XP, Windows Vista, Windows 7, Windows 8, and Windows 10 in both 32-bit and 64-bit versions.

6.2. Developer Options

Android includes support for developers to configure application development-related settings. Device implementations MUST honor the android.settings.APPLICATION_DEVELOPMENT_SETTINGS intent to show application development-related settings The upstream Android implementation hides the Developer Options menu by default and enables users to launch Developer Options after pressing seven (7) times on the **Settings** > **About Device** > **Build Number** menu item. Device implementations MUST provide a consistent experience for Developer Options. Specifically, device implementations MUST hide Developer Options by default and MUST provide a mechanism to enable Developer Options that is consistent with the upstream Android implementation.

Android Automotive implementations MAY limit access to the Developer Options menu by visually hiding or disabling the menu when the vehicle is in motion.

7. Hardware Compatibility

If a device includes a particular hardware component that has a corresponding API for third-party developers, the device implementation MUST implement that API as described in the Android SDK documentation. If an API in the SDK interacts with a hardware component that is stated to be optional and the device implementation does not possess that component:

- Complete class definitions (as documented by the SDK) for the component APIs MUST still be presented.
- The API's behaviors MUST be implemented as no-ops in some reasonable fashion.
- API methods MUST return null values where permitted by the SDK documentation.
- API methods MUST return no-op implementations of classes where null values are not permitted by the SDK documentation.
- API methods MUST NOT throw exceptions not documented by the SDK documentation.

A typical example of a scenario where these requirements apply is the telephony API: Even on non-phone devices, these APIs must be implemented as reasonable no-ops.

Device implementations MUST consistently report accurate hardware configuration information via the getSystemAvailableFeatures() and hasSystemFeature(String) methods on the android.content.pm.PackageManager class for the same build fingerprint.

7.1. Display and Graphics

Android includes facilities that automatically adjust application assets and UI layouts appropriately for the device to ensure that third-party applications run well on a variety of hardware configurations. Devices MUST properly implement these APIs and behaviors, as detailed in this section.

The units referenced by the requirements in this section are defined as follows:

- **physical diagonal size** . The distance in inches between two opposing corners of the illuminated portion of the display.
- **dots per inch (dpi)** . The number of pixels encompassed by a linear horizontal or vertical span of 1". Where dpi values are listed, both horizontal and vertical dpi must fall within the range.
- **aspect ratio** . The ratio of the pixels of the longer dimension to the shorter dimension of the screen. For example, a display of 480x854 pixels would be 854/480 = 1.779, or roughly "16:9".
- **density-independent pixel (dp)** . The virtual pixel unit normalized to a 160 dpi screen, calculated as: pixels = dps * (density/160).

7.1.1. Screen Configuration

7.1.1.1. Screen Size

Android Watch devices (detailed in section 2) MAY have smaller screen sizes as described in this section.

The Android UI framework supports a variety of different screen sizes, and allows applications to query the device screen size (aka "screen layout") via android.content.res.Configuration.screenLayout with the SCREENLAYOUT_SIZE_MASK. Device implementations MUST report the correct screen size as defined in the Android SDK documentation and determined by the upstream Android platform. Specifically, device implementations MUST report the correct screen size according to the following logical density-independent pixel (dp) screen dimensions.

- Devices MUST have screen sizes of at least 426 dp x 320 dp ('small'), unless it is an Android Watch device.
- Devices that report screen size 'normal' MUST have screen sizes of at least 480 dp x 320 dp.
- Devices that report screen size 'large' MUST have screen sizes of at least 640 dp x 480 dp.
- Devices that report screen size 'xlarge' MUST have screen sizes of at least 960 dp x 720 dp.

In addition:

- Android Watch devices MUST have a screen with the physical diagonal size in the range from 1.1 to 2.5 inches.
- Android Automotive devices MUST have a screen with the physical diagonal size greater than or equal to 6 inches.
- Android Automotive devices MUST have a screen size of at least 750 dp x 480 dp.
- Other types of Android device implementations, with a physically integrated screen, MUST have a screen at least 2.5 inches in physical diagonal size.

Devices MUST NOT change their reported screen size at any time.

Applications optionally indicate which screen sizes they support via the <supports-screens> attribute in the AndroidManifest.xml file. Device implementations MUST correctly honor applications' stated support for small, normal, large, and xlarge screens, as described in the Android SDK documentation.

7.1.1.2. Screen Aspect Ratio

Android Watch devices MAY have an aspect ratio of 1.0 (1:1).

The screen aspect ratio MUST be a value from 1.3333 (4:3) to 1.86 (roughly 16:9), but Android Watch devices MAY have an aspect ratio of 1.0 (1:1) because such a device implementation will use a UI_MODE_TYPE_WATCH as the android.content.res.Configuration.uiMode.

7.1.1.3. Screen Density

The Android UI framework defines a set of standard logical densities to help application developers target application resources. Device implementations MUST report only one of the following logical Android framework densities through the android.util.DisplayMetrics APIs, and MUST execute applications at this standard density and MUST NOT change the value at at any time for the default display.

- 120 dpi (ldpi)
- 160 dpi (mdpi)
- 213 dpi (tvdpi)
- 240 dpi (hdpi)
- 280 dpi (280dpi)
- 320 dpi (xhdpi)
- 360 dpi (360dpi)
- 400 dpi (400dpi)
- 420 dpi (420dpi)
- 480 dpi (xxhdpi)
- 560 dpi (560dpi)
- 640 dpi (xxxhdpi)

Device implementations SHOULD define the standard Android framework density that is numerically closest to the physical density of the screen, unless that logical density pushes the reported screen size below the minimum supported. If the standard Android framework density that is numerically closest to the physical density results in a screen size that is smaller than the smallest supported compatible screen size (320 dp width), device implementations SHOULD report the next lowest standard Android framework density.

Device implementations are STRONGLY RECOMMENDED to provide users a setting to change the display size. If there is an implementation to change the display size of the device, it MUST align with the AOSP implementation as indicated below:

- The display size MUST NOT be scaled any larger than 1.5 times the native density or produce an effective minimum screen dimension smaller than 320dp (equivalent to resource qualifier sw320dp), whichever comes first.
- Display size MUST NOT be scaled any smaller than 0.85 times the native density.
- To ensure good usability and consistent font sizes, it is RECOMMENDED that the following scaling of Native Display options be provided (while complying with the limits specified above)
- Small: 0.85x
- Default: 1x (Native display scale)
- Large: 1.15x
- Larger: 1.3x
- Largest 1.45x

7.1.2. Display Metrics

Device implementations MUST report correct values for all display metrics defined in android.util.DisplayMetrics and MUST report the same values regardless of whether the embedded or external screen is used as the default display.

7.1.3. Screen Orientation

Devices MUST report which screen orientations they support (android.hardware.screen.portrait and/or android.hardware.screen.landscape) and MUST report at least one supported orientation. For example, a device with a fixed orientation landscape screen, such as a television or laptop, SHOULD only report android.hardware.screen.landscape.

Devices that report both screen orientations MUST support dynamic orientation by applications to either portrait or landscape screen orientation. That is, the device must respect the application's request for a specific screen orientation. Device implementations MAY select either portrait or landscape orientation as the default.

Devices MUST report the correct value for the device's current orientation, whenever queried via the android.content.res.Configuration.orientation, android.view.Display.getOrientation(), or other APIs.

Devices MUST NOT change the reported screen size or density when changing orientation.

7.1.4. 2D and 3D Graphics Acceleration

Device implementations MUST support both OpenGL ES 1.0 and 2.0, as embodied and detailed in the Android SDK documentations. Device implementations SHOULD support OpenGL ES 3.0, 3.1, or 3.2 on devices capable of supporting it. Device implementations MUST also support Android RenderScript, as detailed in the Android SDK documentation.

Device implementations MUST also correctly identify themselves as supporting OpenGL ES 1.0, OpenGL ES 2.0, OpenGL ES 3.0, OpenGL 3.1, or OpenGL 3.2. That is:

- The managed APIs (such as via the GLES10.getString() method) MUST report support for OpenGL ES 1.0 and OpenGL ES 2.0.
- The native C/C++ OpenGL APIs (APIs available to apps via libGLES_v1CM.so, libGLES_v2.so, or libEGL.so) MUST report support for OpenGL ES 1.0 and OpenGL ES 2.0.
- Device implementations that declare support for OpenGL ES 3.0, 3.1, or 3.2 MUST support the corresponding managed APIs and include support for native C/C++ APIs. On device implementations that declare support for OpenGL ES 3.0, 3.1, or 3.2 libGLESv2.so MUST export the corresponding function symbols in addition to the OpenGL ES 2.0 function symbols.

Android provides an OpenGL ES extension pack with Java interfaces and native support for advanced graphics functionality such as tessellation and the ASTC texture compression format. Android device implementations MUST support the extension pack if the device supports OpenGL ES 3.2 and MAY support it otherwise. If the extension pack is supported in its entirety, the device MUST identify the support through the android.hardware.opengles.aep feature flag.

Also, device implementations MAY implement any desired OpenGL ES extensions. However, device implementations MUST report via the OpenGL ES managed and native APIs all extension strings that they do support, and conversely MUST NOT report extension strings that they do not support.

Note that Android includes support for applications to optionally specify that they require specific OpenGL texture compression formats. These formats are typically vendor-specific. Device implementations are not required by Android to implement any specific texture compression format. However, they SHOULD accurately report any texture compression formats that they do support, via the getString() method in the OpenGL API.

Android includes a mechanism for applications to declare that they want to enable hardware acceleration for 2D graphics at the Application, Activity, Window, or View level through the use of a manifest tag android:hardwareAccelerated or direct API calls.

Device implementations MUST enable hardware acceleration by default, and MUST disable hardware acceleration if the developer so requests by setting android:hardwareAccelerated="false" or disabling hardware acceleration directly through the Android View APIs.

In addition, device implementations MUST exhibit behavior consistent with the Android SDK documentation on hardware acceleration.

Android includes a TextureView object that lets developers directly integrate hardware-accelerated OpenGL ES textures as rendering targets in a UI hierarchy. Device implementations MUST support the TextureView API, and MUST exhibit consistent behavior with the upstream Android implementation.

Android includes support for EGL_ANDROID_RECORDABLE, an EGLConfig attribute that indicates whether the EGLConfig supports rendering to an ANativeWindow that records images to a video. Device implementations MUST support EGL_ANDROID_RECORDABLE extension.

7.1.5. Legacy Application Compatibility Mode

Android specifies a “compatibility mode” in which the framework operates in a 'normal' screen size equivalent (320dp width) mode for the benefit of legacy applications not developed for old versions of Android that pre-date screen-size independence.

- Android Automotive does not support legacy compatibility mode.
- All other device implementations MUST include support for legacy application compatibility mode as implemented by the upstream Android open source code. That is, device implementations MUST NOT alter the triggers or thresholds at which compatibility mode is activated, and MUST NOT alter the behavior of the compatibility mode itself.

7.1.6. Screen Technology

The Android platform includes APIs that allow applications to render rich graphics to the display. Devices MUST support all of these APIs as defined by the Android SDK unless specifically allowed in this document.

- Devices MUST support displays capable of rendering 16-bit color graphics and SHOULD support displays capable of 24-bit color graphics.
- Devices MUST support displays capable of rendering animations.
- The display technology used MUST have a pixel aspect ratio (PAR) between 0.9 and 1.15. That is, the pixel aspect ratio MUST be near square (1.0) with a 10 ~ 15% tolerance.

7.1.7. Secondary Displays

Android includes support for secondary display to enable media sharing capabilities and developer APIs for accessing external displays. If a device supports an external display either via a wired, wireless, or an embedded additional display connection then the device implementation MUST implement the display manager API as described in the Android SDK documentation.

7.2. Input Devices

Devices MUST support a touchscreen or meet the requirements listed in 7.2.2 for non-touch navigation.

7.2.1. Keyboard

Android Watch and Android Automotive implementations MAY implement a soft keyboard. All other device implementations MUST implement a soft keyboard and:

Device implementations:

- MUST include support for the Input Management Framework (which allows third-party developers to create Input Method Editors—i.e. soft keyboard) as detailed at http://developer.android.com.
- MUST provide at least one soft keyboard implementation (regardless of whether a hard keyboard is present) except for Android Watch devices where the screen size makes it less reasonable to have a soft keyboard.
- MAY include additional soft keyboard implementations.
- MAY include a hardware keyboard.
- MUST NOT include a hardware keyboard that does not match one of the formats specified in android.content.res.Configuration.keyboard (QWERTY or 12-key).

7.2.2. Non-touch Navigation

Android Television devices MUST support D-pad.

Device implementations:

- MAY omit a non-touch navigation option (trackball, d-pad, or wheel) if the device implementation is not an Android Television device.
- MUST report the correct value for android.content.res.Configuration.navigation.
- MUST provide a reasonable alternative user interface mechanism for the selection and editing of text, compatible with Input Management Engines. The upstream Android open source implementation includes a selection mechanism suitable for use with devices that lack non-touch navigation inputs.

7.2.3. Navigation Keys

The availability and visibility requirement of the Home, Recents, and Back functions differ between device types as described in this section.

The Home, Recents, and Back functions (mapped to the key events KEYCODE_HOME, KEYCODE_APP_SWITCH, KEYCODE_BACK, respectively) are essential to the Android navigation paradigm and therefore:

- Android Handheld device implementations MUST provide the Home, Recents, and Back functions.
- Android Television device implementations MUST provide the Home and Back functions.
- Android Watch device implementations MUST have the Home function available to the user, and the Back function except for when it is in UI_MODE_TYPE_WATCH .
- Android Watch device implementations, and no other Android device types, MAY consume the long press event on the key event KEYCODE_BACK and omit it from being sent to the foreground application.
- Android Automotive implementations MUST provide the Home function and MAY provide Back and Recent functions.
- All other types of device implementations MUST provide the Home and Back functions.

These functions MAY be implemented via dedicated physical buttons (such as mechanical or capacitive touch buttons), or MAY be implemented using dedicated software keys on a distinct portion

of the screen, gestures, touch panel, etc. Android supports both implementations. All of these functions MUST be accessible with a single action (e.g. tap, double-click or gesture) when visible.

Recents function, if provided, MUST have a visible button or icon unless hidden together with other navigation functions in full-screen mode. This does not apply to devices upgrading from earlier Android versions that have physical buttons for navigation and no recents key.

The Home and Back functions, if provided, MUST each have a visible button or icon unless hidden together with other navigation functions in full-screen mode or when the uiMode UI_MODE_TYPE_MASK is set to UI_MODE_TYPE_WATCH.

The Menu function is deprecated in favor of action bar since Android 4.0. Therefore the new device implementations shipping with Android 7.1 and later MUST NOT implement a dedicated physical button for the Menu function. Older device implementations SHOULD NOT implement a dedicated physical button for the Menu function, but if the physical Menu button is implemented and the device is running applications with targetSdkVersion > 10, the device implementation:

- MUST display the action overflow button on the action bar when it is visible and the resulting action overflow menu popup is not empty. For a device implementation launched before Android 4.4 but upgrading to Android 7.1, this is RECOMMENDED.
- MUST NOT modify the position of the action overflow popup displayed by selecting the overflow button in the action bar.
- MAY render the action overflow popup at a modified position on the screen when it is displayed by selecting the physical menu button.

For backwards compatibility, device implementations MUST make the Menu function available to applications when targetSdkVersion is less than 10, either by a physical button, a software key, or gestures. This Menu function should be presented unless hidden together with other navigation functions.

Android device implementations supporting the Assist action and/or VoiceInteractionService MUST be able to launch an assist app with a single interaction (e.g. tap, double-click, or gesture) when other navigation keys are visible. It is STRONGLY RECOMMENDED to use long press on home as this interaction. The designated interaction MUST launch the user-selected assist app, in other words the app that implements a VoiceInteractionService, or an activity handling the ACTION_ASSIST intent.

Device implementations MAY use a distinct portion of the screen to display the navigation keys, but if so, MUST meet these requirements:

- Device implementation navigation keys MUST use a distinct portion of the screen, not available to applications, and MUST NOT obscure or otherwise interfere with the portion of the screen available to applications.
- Device implementations MUST make available a portion of the display to applications that meets the requirements defined in section 7.1.1.
- Device implementations MUST display the navigation keys when applications do not specify a system UI mode, or specify SYSTEM_UI_FLAG_VISIBLE.
- Device implementations MUST present the navigation keys in an unobtrusive “low profile” (eg. dimmed) mode when applications specify SYSTEM_UI_FLAG_LOW_PROFILE.
- Device implementations MUST hide the navigation keys when applications specify SYSTEM_UI_FLAG_HIDE_NAVIGATION.

7.2.4. Touchscreen Input

Android Handhelds and Watch Devices MUST support touchscreen input.

Device implementations SHOULD have a pointer input system of some kind (either mouse-like or touch). However, if a device implementation does not support a pointer input system, it MUST NOT report the android.hardware.touchscreen or android.hardware.faketouch feature constant. Device implementations that do include a pointer input system:

- SHOULD support fully independently tracked pointers, if the device input system supports multiple pointers.
- MUST report the value of android.content.res.Configuration.touchscreen corresponding to the type of the specific touchscreen on the device.

Android includes support for a variety of touchscreens, touch pads, and fake touch input devices. Touchscreen based device implementations are associated with a display such that the user has the impression of directly manipulating items on screen. Since the user is directly touching the screen, the system does not require any additional affordances to indicate the objects being manipulated. In contrast, a fake touch interface provides a user input system that approximates a subset of touchscreen capabilities. For example, a mouse or remote control that drives an on-screen cursor approximates touch, but requires the user to first point or focus then click. Numerous input devices like the mouse, trackpad, gyro-based air mouse, gyro-pointer, joystick, and multi-touch trackpad can support fake touch interactions. Android includes the feature constant android.hardware.faketouch, which corresponds to a high-fidelity non-touch (pointer-based) input device such as a mouse or trackpad that can adequately emulate touch-based input (including basic gesture support), and indicates that the device supports an emulated subset of touchscreen functionality. Device implementations that declare the fake touch feature MUST meet the fake touch requirements in section 7.2.5.

Device implementations MUST report the correct feature corresponding to the type of input used. Device implementations that include a touchscreen (single-touch or better) MUST report the platform feature constant android.hardware.touchscreen. Device implementations that report the platform feature constant android.hardware.touchscreen MUST also report the platform feature constant android.hardware.faketouch. Device implementations that do not include a touchscreen (and rely on a pointer device only) MUST NOT report any touchscreen feature, and MUST report only android.hardware.faketouch if they meet the fake touch requirements in section 7.2.5.

7.2.5. Fake Touch Input

Device implementations that declare support for android.hardware.faketouch:

- MUST report the absolute X and Y screen positions of the pointer location and display a visual pointer on the screen.
- MUST report touch event with the action code that specifies the state change that occurs on the pointer going down or up on the screen.
- MUST support pointer down and up on an object on the screen, which allows users to emulate tap on an object on the screen.
- MUST support pointer down, pointer up, pointer down then pointer up in the same place on an object on the screen within a time threshold, which allows users to emulate doubletap on an object on the screen.
- MUST support pointer down on an arbitrary point on the screen, pointer move to any other arbitrary point on the screen, followed by a pointer up, which allows users to emulate a touch drag.
- MUST support pointer down then allow users to quickly move the object to a different position on the screen and then pointer up on the screen, which allows users to fling an object on the screen.

Devices that declare support for android.hardware.faketouch.multitouch.distinct MUST meet the requirements for faketouch above, and MUST also support distinct tracking of two or more independent pointer inputs.

7.2.6. Game Controller Support

Android Television device implementations MUST support button mappings for game controllers as listed below. The upstream Android implementation includes implementation for game controllers that satisfies this requirement.

7.2.6.1. Button Mappings

Android Television device implementations MUST support the following key mappings:

Button	HID Usage [2]	Android Button
A [1]	0x09 0x0001	KEYCODE_BUTTON_A (96)
B [1]	0x09 0x0002	KEYCODE_BUTTON_B (97)
X [1]	0x09 0x0004	KEYCODE_BUTTON_X (99)
Y [1]	0x09 0x0005	KEYCODE_BUTTON_Y (100)
D-pad up [1] D-pad down [1]	0x01 0x0039 [3]	AXIS_HAT_Y [4]
D-pad left 1 D-pad right [1]	0x01 0x0039 [3]	AXIS_HAT_X [4]
Left shoulder button [1]	0x09 0x0007	KEYCODE_BUTTON_L1 (102)
Right shoulder button [1]	0x09 0x0008	KEYCODE_BUTTON_R1 (103)
Left stick click [1]	0x09 0x000E	KEYCODE_BUTTON_THUMBL (106)
Right stick click [1]	0x09 0x000F	KEYCODE_BUTTON_THUMBR (107)
Home [1]	0x0c 0x0223	KEYCODE_HOME (3)
Back [1]	0x0c 0x0224	KEYCODE_BACK (4)

1 KeyEvent

2 The above HID usages must be declared within a Game pad CA (0x01 0x0005).

3 This usage must have a Logical Minimum of 0, a Logical Maximum of 7, a Physical Minimum of 0, a Physical Maximum of 315, Units in Degrees, and a Report Size of 4. The logical value is defined to be the clockwise rotation away from the vertical axis; for example, a logical value of 0 represents no rotation and the up button being pressed, while a logical value of 1 represents a rotation of 45 degrees and both the up and left keys being pressed.

4 MotionEvent

Analog Controls [1]	HID Usage	Android Button
Left Trigger	0x02 0x00C5	AXIS_LTRIGGER
Right Trigger	0x02 0x00C4	AXIS_RTRIGGER
Left Joystick	0x01 0x0030 0x01 0x0031	AXIS_X AXIS_Y
Right Joystick	0x01 0x0032 0x01 0x0035	AXIS_Z AXIS_RZ

1 MotionEvent

7.2.7. Remote Control

Android Television device implementations SHOULD provide a remote control to allow users to access the TV interface. The remote control MAY be a physical remote or can be a software-based remote that is accessible from a mobile phone or tablet. The remote control MUST meet the requirements defined below.

- **Search affordance** . Device implementations MUST fire KEYCODE_SEARCH when the user invokes voice search either on the physical or software-based remote.
- **Navigation** . All Android Television remotes MUST include Back, Home, and Select buttons and support for D-pad events .

7.3. Sensors

Android includes APIs for accessing a variety of sensor types. Devices implementations generally MAY omit these sensors, as provided for in the following subsections. If a device includes a particular sensor type that has a corresponding API for third-party developers, the device implementation MUST implement that API as described in the Android SDK documentation and the Android Open Source documentation on sensors. For example, device implementations:

- MUST accurately report the presence or absence of sensors per the android.content.pm.PackageManager class.
- MUST return an accurate list of supported sensors via the SensorManager.getSensorList() and similar methods.
- MUST behave reasonably for all other sensor APIs (for example, by returning true or false as appropriate when applications attempt to register listeners, not calling sensor listeners when the corresponding sensors are not present; etc.).
- MUST report all sensor measurements using the relevant International System of Units (metric) values for each sensor type as defined in the Android SDK documentation.
- SHOULD report the event time in nanoseconds as defined in the Android SDK documentation, representing the time the event happened and synchronized with the SystemClock.elapsedRealtimeNano() clock. Existing and new Android devices are **STRONGLY RECOMMENDED** to meet these requirements so they will be able to upgrade to the future platform releases where this might become a REQUIRED component. The synchronization error SHOULD be below 100 milliseconds.
- MUST report sensor data with a maximum latency of 100 milliseconds + 2 * sample_time for the case of a sensor streamed with a minimum required latency of 5 ms + 2 * sample_time when the application processor is active. This delay does not include any filtering delays.
- MUST report the first sensor sample within 400 milliseconds + 2 * sample_time of the sensor being activated. It is acceptable for this sample to have an accuracy of 0.

The list above is not comprehensive; the documented behavior of the Android SDK and the Android Open Source Documentations on sensors is to be considered authoritative.

Some sensor types are composite, meaning they can be derived from data provided by one or more other sensors. (Examples include the orientation sensor and the linear acceleration sensor.) Device implementations SHOULD implement these sensor types, when they include the prerequisite physical sensors as described in sensor types. If a device implementation includes a composite sensor it MUST implement the sensor as described in the Android Open Source documentation on composite sensors.

Some Android sensors support a “continuous” trigger mode, which returns data continuously. For any API indicated by the Android SDs documentation to be a continuous sensor, device implementations MUST continuously provide periodic data samples that SHOULD have a jitter below 3%, where jitter is defined as the standard deviation of the difference of the reported timestamp values between consecutive events.

Note that the device implementations MUST ensure that the sensor event stream MUST NOT prevent the device CPU from entering a suspend state or waking up from a suspend state.

Finally, when several sensors are activated, the power consumption SHOULD NOT exceed the sum of the individual sensor's reported power consumption.

7.3.1. Accelerometer

Device implementations SHOULD include a 3-axis accelerometer. Android Handheld devices, Android Automotive implementations, and Android Watch devices are STRONGLY RECOMMENDED to include this sensor. If a device implementation does include a 3-axis accelerometer, it:

- MUST implement and report TYPE_ACCELEROMETER sensor.
- MUST be able to report events up to a frequency of at least 50 Hz for Android Watch devices as such devices have a stricter power constraint and 100 Hz for all other device types.
- SHOULD report events up to at least 200 Hz.
- MUST comply with the Android sensor coordinate system as detailed in the Android APIs. Android Automotive implementations MUST comply with the Android car sensor coordinate system.
- MUST be capable of measuring from freefall up to four times the gravity (4g) or more on any axis.
- MUST have a resolution of at least 12-bits and SHOULD have a resolution of at least 16-bits.
- SHOULD be calibrated while in use if the characteristics changes over the life cycle and compensated, and preserve the compensation parameters between device reboots.
- SHOULD be temperature compensated.
- MUST have a standard deviation no greater than 0.05 m/s^, where the standard deviation should be calculated on a per axis basis on samples collected over a period of at least 3 seconds at the fastest sampling rate.
- SHOULD implement the TYPE_SIGNIFICANT_MOTION, TYPE_TILT_DETECTOR, TYPE_STEP_DETECTOR, TYPE_STEP_COUNTER composite sensors as described in the Android SDK document. Existing and new Android devices are **STRONGLY RECOMMENDED** to implement the TYPE_SIGNIFICANT_MOTION composite sensor. If any of these sensors are implemented, the sum of their power consumption MUST always be less than 4 mW and SHOULD each be below 2 mW and 0.5 mW for when the device is in a dynamic or static condition.
- If a gyroscope sensor is included, MUST implement the TYPE_GRAVITY and TYPE_LINEAR_ACCELERATION composite sensors and SHOULD implement the TYPE_GAME_ROTATION_VECTOR composite sensor. Existing and new Android devices are STRONGLY RECOMMENDED to implement the TYPE_GAME_ROTATION_VECTOR sensor.
- MUST implement a TYPE_ROTATION_VECTOR composite sensor, if a gyroscope sensor and a magnetometer sensor is also included.

7.3.2. Magnetometer

Device implementations SHOULD include a 3-axis magnetometer (compass). If a device does include a 3-axis magnetometer, it:

- MUST implement the TYPE_MAGNETIC_FIELD sensor and SHOULD also implement TYPE_MAGNETIC_FIELD_UNCALIBRATED sensor. Existing and new Android devices are STRONGLY RECOMMENDED to implement the TYPE_MAGNETIC_FIELD_UNCALIBRATED sensor.

- MUST be able to report events up to a frequency of at least 10 Hz and SHOULD report events up to at least 50 Hz.
- MUST comply with the Android sensor coordinate system as detailed in the Android APIs.
- MUST be capable of measuring between -900 μT and +900 μT on each axis before saturating.
- MUST have a hard iron offset value less than 700 μT and SHOULD have a value below 200 μT, by placing the magnetometer far from dynamic (current-induced) and static (magnet-induced) magnetic fields.
- MUST have a resolution equal or denser than 0.6 μT and SHOULD have a resolution equal or denser than 0.2 μT.
- SHOULD be temperature compensated.
- MUST support online calibration and compensation of the hard iron bias, and preserve the compensation parameters between device reboots.
- MUST have the soft iron compensation applied—the calibration can be done either while in use or during the production of the device.
- SHOULD have a standard deviation, calculated on a per axis basis on samples collected over a period of at least 3 seconds at the fastest sampling rate, no greater than 0.5 μT.
- MUST implement a TYPE_ROTATION_VECTOR composite sensor, if an accelerometer sensor and a gyroscope sensor is also included.
- MAY implement the TYPE_GEOMAGNETIC_ROTATION_VECTOR sensor if an accelerometer sensor is also implemented. However if implemented, it MUST consume less than 10 mW and SHOULD consume less than 3 mW when the sensor is registered for batch mode at 10 Hz.

7.3.3. GPS

Device implementations SHOULD include a GPS/GNSS receiver. If a device implementation does include a GPS/GNSS receiver and reports the capability to applications through the android.hardware.location.gps feature flag:

- It is STRONGLY RECOMMENDED that the device continue to deliver normal GPS/GNSS outputs to applications during an emergency phone call and that location output not be blocked during an emergency phone call.
- It MUST support location outputs at a rate of at least 1 Hz when requested via LocationManager#requestLocationUpdate .
- It MUST be able to determine the location in open-sky conditions (strong signals, negligible multipath, HDOP < 2) within 10 seconds (fast time to first fix), when connected to a 0.5 Mbps or faster data speed internet connection. This requirement is typically met by the use of some form of Assisted or Predicted GPS/GNSS technique to minimize GPS/GNSS lock-on time (Assistance data includes Reference Time, Reference Location and Satellite Ephemeris/Clock).
  - After making such a location calculation, it is STRONGLY RECOMMENDED for the device to be able to determine its location, in open sky, within 10 seconds, when location requests are restarted, up to an hour after the initial location calculation, even when the subsequent request is made without a data connection, and/or after a power cycle.
- In open sky conditions after determining the location, while stationary or moving with less than 1 meter per second squared of acceleration:
  - It MUST be able to determine location within 20 meters, and speed within 0.5 meters per second, at least 95% of the time.
  - It MUST simultaneously track and report via GnssStatus.Callback at least 8 satellites from one constellation.
  - It SHOULD be able to simultaneously track at least 24 satellites, from multiple

constellations (e.g. GPS + at least one of Glonass, Beidou, Galileo).

- It MUST report the GNSS technology generation through the test API ‘getGnssYearOfHardware’.
- It is STRONGLY RECOMMENDED to meet and MUST meet all requirements below if the GNSS technology generation is reported as the year "2016" or newer.
  - It MUST report GPS measurements, as soon as they are found, even if a location calculated from GPS/GNSS is not yet reported.
  - It MUST report GPS pseudoranges and pseudorange rates, that, in open-sky conditions after determining the location, while stationary or moving with less than 0.2 meter per second squared of acceleration, are sufficient to calculate position within 20 meters, and speed within 0.2 meters per second, at least 95% of the time.

Note that while some of the GPS requirements above are stated as STRONGLY RECOMMENDED, the Compatibility Definition for the next major version is expected to change these to a MUST.

7.3.4. Gyroscope

Device implementations SHOULD include a gyroscope (angular change sensor). Devices SHOULD NOT include a gyroscope sensor unless a 3-axis accelerometer is also included. If a device implementation includes a gyroscope, it:

- MUST implement the TYPE_GYROSCOPE sensor and SHOULD also implement TYPE_GYROSCOPE_UNCALIBRATED sensor. Existing and new Android devices are STRONGLY RECOMMENDED to implement the SENSOR_TYPE_GYROSCOPE_UNCALIBRATED sensor.
- MUST be capable of measuring orientation changes up to 1,000 degrees per second.
- MUST be able to report events up to a frequency of at least 50 Hz for Android Watch devices as such devices have a stricter power constraint and 100 Hz for all other device types.
- SHOULD report events up to at least 200 Hz.
- MUST have a resolution of 12-bits or more and SHOULD have a resolution of 16-bits or more.
- MUST be temperature compensated.
- MUST be calibrated and compensated while in use, and preserve the compensation parameters between device reboots.
- MUST have a variance no greater than 1e-7 rad^2 / s^2 per Hz (variance per Hz, or rad^2 / s). The variance is allowed to vary with the sampling rate, but must be constrained by this value. In other words, if you measure the variance of the gyro at 1 Hz sampling rate it should be no greater than 1e-7 rad^2/s^2.
- MUST implement a TYPE_ROTATION_VECTOR composite sensor, if an accelerometer sensor and a magnetometer sensor is also included.
- If an accelerometer sensor is included, MUST implement the TYPE_GRAVITY and TYPE_LINEAR_ACCELERATION composite sensors and SHOULD implement the TYPE_GAME_ROTATION_VECTOR composite sensor. Existing and new Android devices are STRONGLY RECOMMENDED to implement the TYPE_GAME_ROTATION_VECTOR sensor.

7.3.5. Barometer

Device implementations SHOULD include a barometer (ambient air pressure sensor). If a device implementation includes a barometer, it:

- MUST implement and report TYPE_PRESSURE sensor.
- MUST be able to deliver events at 5 Hz or greater.
- MUST have adequate precision to enable estimating altitude.
- MUST be temperature compensated.

7.3.6. Thermometer

Device implementations MAY include an ambient thermometer (temperature sensor). If present, it MUST be defined as SENSOR_TYPE_AMBIENT_TEMPERATURE and it MUST measure the ambient (room) temperature in degrees Celsius.

Device implementations MAY but SHOULD NOT include a CPU temperature sensor. If present, it MUST be defined as SENSOR_TYPE_TEMPERATURE, it MUST measure the temperature of the device CPU, and it MUST NOT measure any other temperature. Note the SENSOR_TYPE_TEMPERATURE sensor type was deprecated in Android 4.0.

For Android Automotive implementations, SENSOR_TYPE_AMBIENT_TEMPERATURE MUST measure the temperature inside the vehicle cabin.

7.3.7. Photometer

Device implementations MAY include a photometer (ambient light sensor).

7.3.8. Proximity Sensor

Device implementations MAY include a proximity sensor. Devices that can make a voice call and indicate any value other than PHONE_TYPE_NONE in getPhoneType SHOULD include a proximity sensor. If a device implementation does include a proximity sensor, it:

- MUST measure the proximity of an object in the same direction as the screen. That is, the proximity sensor MUST be oriented to detect objects close to the screen, as the primary intent of this sensor type is to detect a phone in use by the user. If a device implementation includes a proximity sensor with any other orientation, it MUST NOT be accessible through this API.
- MUST have 1-bit of accuracy or more.

7.3.9. High Fidelity Sensors

Device implementations supporting a set of higher quality sensors that can meet all the requirements listed in this section MUST identify the support through the android.hardware.sensor.hifi_sensors feature flag.

A device declaring android.hardware.sensor.hifi_sensors MUST support all of the following sensor types meeting the quality requirements as below:

- SENSOR_TYPE_ACCELEROMETER
  - MUST have a measurement range between at least -8g and +8g.
  - MUST have a measurement resolution of at least 1024 LSB/G.
  - MUST have a minimum measurement frequency of 12.5 Hz or lower.
  - MUST have a maximum measurement frequency of 400 Hz or higher.
  - MUST have a measurement noise not above 400 uG/√Hz.
  - MUST implement a non-wake-up form of this sensor with a buffering capability of at least 3000 sensor events.
  - MUST have a batching power consumption not worse than 3 mW.
  - SHOULD have a stationary noise bias stability of \<15 μg √Hz from 24hr static

dataset.
    - SHOULD have a bias change vs. temperature of ≤ +/- 1mg / °C.
    - SHOULD have a best-fit line non-linearity of ≤ 0.5%, and sensitivity change vs. temperature of ≤ 0.03%/C°.
- SENSOR_TYPE_GYROSCOPE
    - MUST have a measurement range between at least -1000 and +1000 dps.
    - MUST have a measurement resolution of at least 16 LSB/dps.
    - MUST have a minimum measurement frequency of 12.5 Hz or lower.
    - MUST have a maximum measurement frequency of 400 Hz or higher.
    - MUST have a measurement noise not above 0.014°/s/√Hz.
    - SHOULD have a stationary bias stability of < 0.0002 °/s √Hz from 24-hour static dataset.
    - SHOULD have a bias change vs. temperature of ≤ +/- 0.05 °/ s / °C.
    - SHOULD have a sensitivity change vs. temperature of ≤ 0.02% / °C.
    - SHOULD have a best-fit line non-linearity of ≤ 0.2%.
    - SHOULD have a noise density of ≤ 0.07 °/s/√Hz.
- SENSOR_TYPE_GYROSCOPE_UNCALIBRATED with the same quality requirements as SENSOR_TYPE_GYROSCOPE.
- SENSOR_TYPE_GEOMAGNETIC_FIELD
    - MUST have a measurement range between at least -900 and +900 uT.
    - MUST have a measurement resolution of at least 5 LSB/uT.
    - MUST have a minimum measurement frequency of 5 Hz or lower.
    - MUST have a maximum measurement frequency of 50 Hz or higher.
    - MUST have a measurement noise not above 0.5 uT.
- SENSOR_TYPE_MAGNETIC_FIELD_UNCALIBRATED with the same quality requirements as SENSOR_TYPE_GEOMAGNETIC_FIELD and in addition:
    - MUST implement a non-wake-up form of this sensor with a buffering capability of at least 600 sensor events.
- SENSOR_TYPE_PRESSURE
    - MUST have a measurement range between at least 300 and 1100 hPa.
    - MUST have a measurement resolution of at least 80 LSB/hPa.
    - MUST have a minimum measurement frequency of 1 Hz or lower.
    - MUST have a maximum measurement frequency of 10 Hz or higher.
    - MUST have a measurement noise not above 2 Pa/√Hz.
    - MUST implement a non-wake-up form of this sensor with a buffering capability of at least 300 sensor events.
    - MUST have a batching power consumption not worse than 2 mW.
- SENSOR_TYPE_GAME_ROTATION_VECTOR
    - MUST implement a non-wake-up form of this sensor with a buffering capability of at least 300 sensor events.
    - MUST have a batching power consumption not worse than 4 mW.
- SENSOR_TYPE_SIGNIFICANT_MOTION
    - MUST have a power consumption not worse than 0.5 mW when device is static and 1.5 mW when device is moving.
- SENSOR_TYPE_STEP_DETECTOR
    - MUST implement a non-wake-up form of this sensor with a buffering capability of at least 100 sensor events.
    - MUST have a power consumption not worse than 0.5 mW when device is static and 1.5 mW when device is moving.

    - MUST have a batching power consumption not worse than 4 mW.
- SENSOR_TYPE_STEP_COUNTER
    - MUST have a power consumption not worse than 0.5 mW when device is static and 1.5 mW when device is moving.
- SENSOR_TILT_DETECTOR
    - MUST have a power consumption not worse than 0.5 mW when device is static and 1.5 mW when device is moving.

Also such a device MUST meet the following sensor subsystem requirements:

- The event timestamp of the same physical event reported by the Accelerometer, Gyroscope sensor and Magnetometer MUST be within 2.5 milliseconds of each other.
- The Gyroscope sensor event timestamps MUST be on the same time base as the camera subsystem and within 1 milliseconds of error.
- High Fidelity sensors MUST deliver samples to applications within 5 milliseconds from the time when the data is available on the physical sensor to the application.
- The power consumption MUST not be higher than 0.5 mW when device is static and 2.0 mW when device is moving when any combination of the following sensors are enabled:
    - SENSOR_TYPE_SIGNIFICANT_MOTION
    - SENSOR_TYPE_STEP_DETECTOR
    - SENSOR_TYPE_STEP_COUNTER
    - SENSOR_TILT_DETECTORS

Note that all power consumption requirements in this section do not include the power consumption of the Application Processor. It is inclusive of the power drawn by the entire sensor chain—the sensor, any supporting circuitry, any dedicated sensor processing system, etc.

The following sensor types MAY also be supported on a device implementation declaring android.hardware.sensor.hifi_sensors, but if these sensor types are present they MUST meet the following minimum buffering capability requirement:

- SENSOR_TYPE_PROXIMITY: 100 sensor events

7.3.10. Fingerprint Sensor

Device implementations with a secure lock screen SHOULD include a fingerprint sensor. If a device implementation includes a fingerprint sensor and has a corresponding API for third-party developers, it:

- MUST declare support for the android.hardware.fingerprint feature.
- MUST fully implement the corresponding API as described in the Android SDK documentation.
- MUST have a false acceptance rate not higher than 0.002%.
- Is STRONGLY RECOMMENDED to have a false rejection rate of less than 10%, as measured on the device
- Is STRONGLY RECOMMENDED to have a latency below 1 second, measured from when the fingerprint sensor is touched until the screen is unlocked, for one enrolled finger.
- MUST rate limit attempts for at least 30 seconds after five false trials for fingerprint verification.
- MUST have a hardware-backed keystore implementation, and perform the fingerprint matching in a Trusted Execution Environment (TEE) or on a chip with a secure channel to the TEE.
- MUST have all identifiable fingerprint data encrypted and cryptographically authenticated

such that they cannot be acquired, read or altered outside of the Trusted Execution Environment (TEE) as documented in the implementation guidelines on the Android Open Source Project site.
- MUST prevent adding a fingerprint without first establishing a chain of trust by having the user confirm existing or add a new device credential (PIN/pattern/password) using the TEE as implemented in the Android Open Source project.
- MUST NOT enable 3rd-party applications to distinguish between individual fingerprints.
- MUST honor the DevicePolicyManager.KEYGUARD_DISABLE_FINGERPRINT flag.
- MUST, when upgraded from a version earlier than Android 6.0, have the fingerprint data securely migrated to meet the above requirements or removed.
- SHOULD use the Android Fingerprint icon provided in the Android Open Source Project.

7.3.11. Android Automotive-only sensors

Automotive-specific sensors are defined in the android.car.CarSensorManager API .

7.3.11.1. Current Gear

Android Automotive implementations SHOULD provide current gear as SENSOR_TYPE_GEAR.

7.3.11.2. Day Night Mode

Android Automotive implementations MUST support day/night mode defined as SENSOR_TYPE_NIGHT. The value of this flag MUST be consistent with dashboard day/night mode and SHOULD be based on ambient light sensor input. The underlying ambient light sensor MAY be the same as Photometer.

7.3.11.3. Driving Status

Android Automotive implementations MUST support driving status defined as SENSOR_TYPE_DRIVING_STATUS, with a default value of DRIVE_STATUS_UNRESTRICTED when the vehicle is fully stopped and parked. It is the responsibility of device manufacturers to configure SENSOR_TYPE_DRIVING_STATUS in compliance with all laws and regulations that apply to markets where the product is shipping.

7.3.11.4. Wheel Speed

Android Automotive implementations MUST provide vehicle speed defined as SENSOR_TYPE_CAR_SPEED.

7.3.12. Pose Sensor

Device implementations MAY support pose sensor with 6 degrees of freedom. Android Handheld devices are RECOMMENDED to support this sensor. If a device implementation does support pose sensor with 6 degrees of freedom, it:

- MUST implement and report TYPE_POSE_6DOF sensor.
- MUST be more accurate than the rotation vector alone.

7.4. Data Connectivity

7.4.1. Telephony

“Telephony” as used by the Android APIs and this document refers specifically to hardware related to placing voice calls and sending SMS messages via a GSM or CDMA network. While these voice calls may or may not be packet-switched, they are for the purposes of Android considered independent of any data connectivity that may be implemented using the same network. In other words, the Android “telephony” functionality and APIs refer specifically to voice calls and SMS. For instance, device implementations that cannot place calls or send/receive SMS messages MUST NOT report the android.hardware.telephony feature or any subfeatures, regardless of whether they use a cellular network for data connectivity.

Android MAY be used on devices that do not include telephony hardware. That is, Android is compatible with devices that are not phones. However, if a device implementation does include GSM or CDMA telephony, it MUST implement full support for the API for that technology. Device implementations that do not include telephony hardware MUST implement the full APIs as no-ops.

7.4.1.1. Number Blocking Compatibility

Android Telephony device implementations MUST include number blocking support and:

- MUST fully implement BlockedNumberContract and the corresponding API as described in the SDK documentation.
- MUST block all calls and messages from a phone number in 'BlockedNumberProvider' without any interaction with apps. The only exception is when number blocking is temporarily lifted as described in the SDK documentation.
- MUST NOT write to the [platform call log provider] (http://developer.android.com/reference/android/provider/CallLog.html) for a blocked call.
- MUST NOT write to the [Telephony provider] (http://developer.android.com/reference/android/provider/Telephony.html) for a blocked message.
- MUST implement a blocked numbers management UI, which is opened with the intent returned by TelecomManager.createManageBlockedNumbersIntent() method.
- MUST NOT allow secondary users to view or edit the blocked numbers on the device as the Android platform assumes the primary user to have full control of the telephony services, a single instance, on the device. All blocking related UI MUST be hidden for secondary users and the blocked list MUST still be respected.
- SHOULD migrate the blocked numbers into the provider when a device updates to Android 7.0.

7.4.2. IEEE 802.11 (Wi-Fi)

All Android device implementations SHOULD include support for one or more forms of 802.11. If a device implementation does include support for 802.11 and exposes the functionality to a third-party application, it MUST implement the corresponding Android API and:

- MUST report the hardware feature flag android.hardware.wifi.
- MUST implement the multicast API as described in the SDK documentation.
- MUST support multicast DNS (mDNS) and MUST NOT filter mDNS packets (224.0.0.251) at any time of operation including:
  - Even when the screen is not in an active state.
  - For Android Television device implementations, even when in standby power states.

7.4.2.1. Wi-Fi Direct

Device implementations SHOULD include support for Wi-Fi Direct (Wi-Fi peer-to-peer). If a device implementation does include support for Wi-Fi Direct, it MUST implement the corresponding Android API as described in the SDK documentation. If a device implementation includes support for Wi-Fi Direct, then it:

- MUST report the hardware feature android.hardware.wifi.direct.
- MUST support regular Wi-Fi operation.
- SHOULD support concurrent Wi-Fi and Wi-Fi Direct operation.

7.4.2.2. Wi-Fi Tunneled Direct Link Setup

Device implementations SHOULD include support for Wi-Fi Tunneled Direct Link Setup (TDLS) as described in the Android SDK Documentation. If a device implementation does include support for TDLS and TDLS is enabled by the WiFiManager API, the device:

- SHOULD use TDLS only when it is possible AND beneficial.
- SHOULD have some heuristic and NOT use TDLS when its performance might be worse than going through the Wi-Fi access point.

7.4.3. Bluetooth

Android Watch implementations MUST support Bluetooth. Android Television implementations MUST support Bluetooth and Bluetooth LE. Android Automotive implementations MUST support Bluetooth and SHOULD support Bluetooth LE.

Device implementations that support android.hardware.vr.high_performance feature MUST support Bluetooth 4.2 and Bluetooth LE Data Length Extension.

Android includes support for Bluetooth and Bluetooth Low Energy. Device implementations that include support for Bluetooth and Bluetooth Low Energy MUST declare the relevant platform features (android.hardware.bluetooth and android.hardware.bluetooth_le respectively) and implement the platform APIs. Device implementations SHOULD implement relevant Bluetooth profiles such as A2DP, AVCP, OBEX, etc. as appropriate for the device.

Android Automotive implementations SHOULD support Message Access Profile (MAP). Android Automotive implementations MUST support the following Bluetooth profiles:

- Phone calling over Hands-Free Profile (HFP).
- Media playback over Audio Distribution Profile (A2DP).
- Media playback control over Remote Control Profile (AVRCP).
- Contact sharing using the Phone Book Access Profile (PBAP).

Device implementations including support for Bluetooth Low Energy:

- MUST declare the hardware feature android.hardware.bluetooth_le.
- MUST enable the GATT (generic attribute profile) based Bluetooth APIs as described in the SDK documentation and android.bluetooth.
- are STRONGLY RECOMMENDED to implement a Resolvable Private Address (RPA) timeout no longer than 15 minutes and rotate the address at timeout to protect user privacy.
- SHOULD support offloading of the filtering logic to the bluetooth chipset when implementing the ScanFilter API, and MUST report the correct value of where the filtering logic is implemented whenever queried via the

android.bluetooth.BluetoothAdapter.isOffloadedFilteringSupported() method.

- SHOULD support offloading of the batched scanning to the bluetooth chipset, but if not supported, MUST report 'false' whenever queried via the android.bluetooth.BluetoothAdapter.isOffloadedScanBatchingSupported() method.
- SHOULD support multi advertisement with at least 4 slots, but if not supported, MUST report 'false' whenever queried via the android.bluetooth.BluetoothAdapter.isMultipleAdvertisementSupported() method.

7.4.4. Near-Field Communications

Device implementations SHOULD include a transceiver and related hardware for Near-Field Communications (NFC). If a device implementation does include NFC hardware and plans to make it available to third-party apps, then it:

- MUST report the android.hardware.nfc feature from the android.content.pm.PackageManager.hasSystemFeature() method.
- MUST be capable of reading and writing NDEF messages via the following NFC standards:
  - MUST be capable of acting as an NFC Forum reader/writer (as defined by the NFC Forum technical specification NFCForum-TS-DigitalProtocol-1.0) via the following NFC standards:
    - NfcA (ISO14443-3A)
    - NfcB (ISO14443-3B)
    - NfcF (JIS X 6319-4)
    - IsoDep (ISO 14443-4)
    - NFC Forum Tag Types 1, 2, 3, 4 (defined by the NFC Forum)
  - STRONGLY RECOMMENDED to be capable of reading and writing NDEF messages as well as raw data via the following NFC standards. Note that while the NFC standards below are stated as STRONGLY RECOMMENDED, the Compatibility Definition for a future version is planned to change these to MUST. These standards are optional in this version but will be required in future versions. Existing and new devices that run this version of Android are very strongly encouraged to meet these requirements now so they will be able to upgrade to the future platform releases.
    - NfcV (ISO 15693)
  - SHOULD be capable of reading the barcode and URL (if encoded) of Thinfilm NFC Barcode products.
  - MUST be capable of transmitting and receiving data via the following peer-to-peer standards and protocols:
    - ISO 18092
    - LLCP 1.2 (defined by the NFC Forum)
    - SDP 1.0 (defined by the NFC Forum)
    - NDEF Push Protocol
    - SNEP 1.0 (defined by the NFC Forum)
  - MUST include support for Android Beam.
  - MUST implement the SNEP default server. Valid NDEF messages received by the default SNEP server MUST be dispatched to applications using the android.nfc.ACTION_NDEF_DISCOVERED intent. Disabling Android Beam in settings MUST NOT disable dispatch of incoming NDEF message.
  - MUST honor the android.settings.NFCSHARING_SETTINGS intent to show NFC sharing settings.
  - MUST implement the NPP server. Messages received by the NPP server

MUST be processed the same way as the SNEP default server.
    - MUST implement a SNEP client and attempt to send outbound P2P NDEF to the default SNEP server when Android Beam is enabled. If no default SNEP server is found then the client MUST attempt to send to an NPP server.
    - MUST allow foreground activities to set the outbound P2P NDEF message using android.nfc.NfcAdapter.setNdefPushMessage, and android.nfc.NfcAdapter.setNdefPushMessageCallback, and android.nfc.NfcAdapter.enableForegroundNdefPush.
    - SHOULD use a gesture or on-screen confirmation, such as 'Touch to Beam', before sending outbound P2P NDEF messages.
    - SHOULD enable Android Beam by default and MUST be able to send and receive using Android Beam, even when another proprietary NFC P2p mode is turned on.
    - MUST support NFC Connection handover to Bluetooth when the device supports Bluetooth Object Push Profile. Device implementations MUST support connection handover to Bluetooth when using android.nfc.NfcAdapter.setBeamPushUris, by implementing the " Connection Handover version 1.2 " and " Bluetooth Secure Simple Pairing Using NFC version1.0 " specs from the NFC Forum. Such an implementation MUST implement the handover LLCP service with service name "urn:nfc:sn:handover" for exchanging the handover request/select records over NFC, and it MUST use the Bluetooth Object Push Profile for the actual Bluetooth data transfer. For legacy reasons (to remain compatible with Android 4.1 devices), the implementation SHOULD still accept SNEP GET requests for exchanging the handover request/select records over NFC. However an implementation itself SHOULD NOT send SNEP GET requests for performing connection handover.
    - MUST poll for all supported technologies while in NFC discovery mode.
    - SHOULD be in NFC discovery mode while the device is awake with the screen active and the lock-screen unlocked.

(Note that publicly available links are not available for the JIS, ISO, and NFC Forum specifications cited above.)

Android includes support for NFC Host Card Emulation (HCE) mode. If a device implementation does include an NFC controller chipset capable of HCE (for NfcA and/or NfcB) and it supports Application ID (AID) routing, then it:

- MUST report the android.hardware.nfc.hce feature constant.
- MUST support NFC HCE APIs as defined in the Android SDK.

If a device implementation does include an NFC controller chipset capable of HCE for NfcF, and it implements the feature for third-party applications, then it:

- MUST report the android.hardware.nfc.hcef feature constant.
- MUST implement the [NfcF Card Emulation APIs] (https://developer.android.com/reference/android/nfc/cardemulation/NfcFCardEmulation.html) as defined in the Android SDK.

Additionally, device implementations MAY include reader/writer support for the following MIFARE technologies.

- MIFARE Classic
- MIFARE Ultralight
- NDEF on MIFARE Classic

Note that Android includes APIs for these MIFARE types. If a device implementation supports MIFARE in the reader/writer role, it:

- MUST implement the corresponding Android APIs as documented by the Android SDK.
- MUST report the feature com.nxp.mifare from the android.content.pm.PackageManager.hasSystemFeature() method. Note that this is not a standard Android feature and as such does not appear as a constant in the android.content.pm.PackageManager class.
- MUST NOT implement the corresponding Android APIs nor report the com.nxp.mifare feature unless it also implements general NFC support as described in this section.

If a device implementation does not include NFC hardware, it MUST NOT declare the android.hardware.nfc feature from the android.content.pm.PackageManager.hasSystemFeature() method, and MUST implement the Android NFC API as a no-op.

As the classes android.nfc.NdefMessage and android.nfc.NdefRecord represent a protocol-independent data representation format, device implementations MUST implement these APIs even if they do not include support for NFC or declare the android.hardware.nfc feature.

7.4.5. Minimum Network Capability

Device implementations MUST include support for one or more forms of data networking. Specifically, device implementations MUST include support for at least one data standard capable of 200Kbit/sec or greater. Examples of technologies that satisfy this requirement include EDGE, HSPA, EV-DO, 802.11g, Ethernet, Bluetooth PAN, etc.

Device implementations where a physical networking standard (such as Ethernet) is the primary data connection SHOULD also include support for at least one common wireless data standard, such as 802.11 (Wi-Fi).

Devices MAY implement more than one form of data connectivity.

Devices MUST include an IPv6 networking stack and support IPv6 communication using the managed APIs, such as java.net.Socket and java.net.URLConnection , as well as the native APIs, such as AF_INET6 sockets. The required level of IPv6 support depends on the network type, as follows:

- Devices that support Wi-Fi networks MUST support dual-stack and IPv6-only operation on Wi-Fi.
- Devices that support Ethernet networks MUST support dual-stack operation on Ethernet.
- Devices that support cellular data SHOULD support IPv6 operation (IPv6-only and possibly dual-stack) on cellular data.
- When a device is simultaneously connected to more than one network (e.g., Wi-Fi and cellular data), it MUST simultaneously meet these requirements on each network to which it is connected.

IPv6 MUST be enabled by default.

In order to ensure that IPv6 communication is as reliable as IPv4, unicast IPv6 packets sent to the device MUST NOT be dropped, even when the screen is not in an active state. Redundant multicast IPv6 packets, such as repeated identical Router Advertisements, MAY be rate-limited in hardware or firmware if doing so is necessary to save power. In such cases, rate-limiting MUST NOT cause the device to lose IPv6 connectivity on any IPv6-compliant network that uses RA lifetimes of at least 180 seconds.

IPv6 connectivity MUST be maintained in doze mode.

7.4.6. Sync Settings

Device implementations MUST have the master auto-sync setting on by default so that the method getMasterSyncAutomatically() returns “true”.

7.4.7. Data Saver

Device implementations with a metered connection are STRONGLY RECOMMENDED to provide the data saver mode.

If a device implementation provides the data saver mode, it:

- MUST support all the APIs in the ConnectivityManager class as described in the SDK documentation
- MUST provide a user interface in the settings, allowing users to add applications to or remove applications from the whitelist.

Conversely if a device implementation does not provide the data saver mode, it:

- MUST return the value RESTRICT_BACKGROUND_STATUS_DISABLED for ConnectivityManager.getRestrictBackgroundStatus()
- MUST not broadcast ConnectivityManager.ACTION_RESTRICT_BACKGROUND_CHANGED
- MUST have an activity that handles the Settings.ACTION_IGNORE_BACKGROUND_DATA_RESTRICTIONS_SETTINGS intent but MAY implement it as a no-op.

7.5. Cameras

Device implementations SHOULD include a rear-facing camera and MAY include a front-facing camera. A rear-facing camera is a camera located on the side of the device opposite the display; that is, it images scenes on the far side of the device, like a traditional camera. A front-facing camera is a camera located on the same side of the device as the display; that is, a camera typically used to image the user, such as for video conferencing and similar applications.

If a device implementation includes at least one camera, it MUST be possible for an application to simultaneously allocate 3 RGBA_8888 bitmaps equal to the size of the images produced by the largest-resolution camera sensor on the device, while camera is open for the purpose of basic preview and still capture.

7.5.1. Rear-Facing Camera

Device implementations SHOULD include a rear-facing camera. If a device implementation includes at least one rear-facing camera, it:

- MUST report the feature flag android.hardware.camera and android.hardware.camera.any.
- MUST have a resolution of at least 2 megapixels.
- SHOULD have either hardware auto-focus or software auto-focus implemented in the camera driver (transparent to application software).
- MAY have fixed-focus or EDOF (extended depth of field) hardware.
- MAY include a flash. If the Camera includes a flash, the flash lamp MUST NOT be lit while an android.hardware.Camera.PreviewCallback instance has been registered on a Camera preview surface, unless the application has explicitly enabled the flash by enabling the FLASH_MODE_AUTO or FLASH_MODE_ON attributes of a Camera.Parameters object. Note that this constraint does not apply to the device’s built-in system camera application,

but only to third-party applications using Camera.PreviewCallback.

7.5.2. Front-Facing Camera

Device implementations MAY include a front-facing camera. If a device implementation includes at least one front-facing camera, it:

- MUST report the feature flag android.hardware.camera.any and android.hardware.camera.front.
- MUST have a resolution of at least VGA (640x480 pixels).
- MUST NOT use a front-facing camera as the default for the Camera API. The camera API in Android has specific support for front-facing cameras and device implementations MUST NOT configure the API to to treat a front-facing camera as the default rear-facing camera, even if it is the only camera on the device.
- MAY include features (such as auto-focus, flash, etc.) available to rear-facing cameras as described in section 7.5.1.
- MUST horizontally reflect (i.e. mirror) the stream displayed by an app in a CameraPreview, as follows:
  - If the device implementation is capable of being rotated by user (such as automatically via an accelerometer or manually via user input), the camera preview MUST be mirrored horizontally relative to the device's current orientation.
  - If the current application has explicitly requested that the Camera display be rotated via a call to the android.hardware.Camera.setDisplayOrientation() method, the camera preview MUST be mirrored horizontally relative to the orientation specified by the application.
  - Otherwise, the preview MUST be mirrored along the device's default horizontal axis.
- MUST mirror the image displayed by the postview in the same manner as the camera preview image stream. If the device implementation does not support postview, this requirement obviously does not apply.
- MUST NOT mirror the final captured still image or video streams returned to application callbacks or committed to media storage.

7.5.3. External Camera

Device implementations MAY include support for an external camera that is not necessarily always connected. If a device includes support for an external camera, it:

- MUST declare the platform feature flag android.hardware.camera.external and android.hardware camera.any .
- MAY support multiple cameras.
- MUST support USB Video Class (UVC 1.0 or higher) if the external camera connects through the USB port.
- SHOULD support video compressions such as MJPEG to enable transfer of high-quality unencoded streams (i.e. raw or independently compressed picture streams).
- MAY support camera-based video encoding. If supported, a simultaneous unencoded / MJPEG stream (QVGA or greater resolution) MUST be accessible to the device implementation.

7.5.4. Camera API Behavior

Android includes two API packages to access the camera, the newer android.hardware.camera2 API

expose lower-level camera control to the app, including efficient zero-copy burst/streaming flows and per-frame controls of exposure, gain, white balance gains, color conversion, denoising, sharpening, and more.

The older API package, android.hardware.Camera, is marked as deprecated in Android 5.0 but as it should still be available for apps to use Android device implementations MUST ensure the continued support of the API as described in this section and in the Android SDK.

Device implementations MUST implement the following behaviors for the camera-related APIs, for all available cameras:

- If an application has never called android.hardware.Camera.Parameters.setPreviewFormat(int), then the device MUST use android.hardware.PixelFormat.YCbCr_420_SP for preview data provided to application callbacks.
- If an application registers an android.hardware.Camera.PreviewCallback instance and the system calls the onPreviewFrame() method when the preview format is YCbCr_420_SP, the data in the byte[] passed into onPreviewFrame() must further be in the NV21 encoding format. That is, NV21 MUST be the default.
- For android.hardware.Camera, device implementations MUST support the YV12 format (as denoted by the android.graphics.ImageFormat.YV12 constant) for camera previews for both front- and rear-facing cameras. (The hardware video encoder and camera may use any native pixel format, but the device implementation MUST support conversion to YV12.)
- For android.hardware.camera2, device implementations must support the android.hardware.ImageFormat.YUV_420_888 and android.hardware.ImageFormat.JPEG formats as outputs through the android.media.ImageReader API.

Device implementations MUST still implement the full Camera API included in the Android SDK documentation, regardless of whether the device includes hardware autofocus or other capabilities. For instance, cameras that lack autofocus MUST still call any registered android.hardware.Camera.AutoFocusCallback instances (even though this has no relevance to a non-autofocus camera.) Note that this does apply to front-facing cameras; for instance, even though most front-facing cameras do not support autofocus, the API callbacks must still be “faked” as described.

Device implementations MUST recognize and honor each parameter name defined as a constant on the android.hardware.Camera.Parameters class, if the underlying hardware supports the feature. If the device hardware does not support a feature, the API must behave as documented. Conversely, device implementations MUST NOT honor or recognize string constants passed to the android.hardware.Camera.setParameters() method other than those documented as constants on the android.hardware.Camera.Parameters. That is, device implementations MUST support all standard Camera parameters if the hardware allows, and MUST NOT support custom Camera parameter types. For instance, device implementations that support image capture using high dynamic range (HDR) imaging techniques MUST support camera parameter Camera.SCENE_MODE_HDR.

Because not all device implementations can fully support all the features of the android.hardware.camera2 API, device implementations MUST report the proper level of support with the android.info.supportedHardwareLevel property as described in the Android SDK and report the appropriate frameworkfeature flags.

Device implementations MUST also declare its Individual camera capabilities of android.hardware.camera2 via the android.request.availableCapabilities property and declare the appropriate feature flags ; a device must define the feature flag if any of its attached camera devices supports the feature.

Device implementations MUST broadcast the Camera.ACTION_NEW_PICTURE intent whenever a new picture is taken by the camera and the entry of the picture has been added to the media store.

Device implementations MUST broadcast the Camera.ACTION_NEW_VIDEO intent whenever a new video is recorded by the camera and the entry of the picture has been added to the media store.

7.5.5. Camera Orientation

Both front- and rear-facing cameras, if present, MUST be oriented so that the long dimension of the camera aligns with the screen's long dimension. That is, when the device is held in the landscape orientation, cameras MUST capture images in the landscape orientation. This applies regardless of the device's natural orientation; that is, it applies to landscape-primary devices as well as portrait-primary devices.

7.6. Memory and Storage

7.6.1. Minimum Memory and Storage

Android Television devices MUST have at least 4GB of non-volatile storage available for application private data.

The memory available to the kernel and userspace on device implementations MUST be at least equal or larger than the minimum values specified by the following table. (See section 7.1.1 for screen size and density definitions.)

Density and screen size	32-bit device	64-bit device
Android Watch devices (due to smaller screens)	416MB	Not applicable
• 280dpi or lower on small/normal screens • mdpi or lower on large screens • ldpi or lower on extra large screens	512MB	816MB
• xhdpi or higher on small/normal screens • hdpi or higher on large screens • mdpi or higher on extra large screens	608MB	944MB
• 400dpi or higher on small/normal screens • xhdpi or higher on large screens • tvdpi or higher on extra large screens	896MB	1280MB
• 560dpi or higher on small/normal screens • 400dpi or higher on large screens • xhdpi or higher on extra large screens	1344MB	1824MB

The minimum memory values MUST be in addition to any memory space already dedicated to hardware components such as radio, video, and so on that is not under the kernel's control.

Device implementations with less than 512MB of memory available to the kernel and userspace, unless an Android Watch, MUST return the value "true" for ActivityManager.isLowRamDevice().

Android Television devices MUST have at least 4GB and other device implementations MUST have at least 3GB of non-volatile storage available for application private data. That is, the /data partition MUST be at least 4GB for Android Television devices and at least 3GB for other device implementations. Device implementations that run Android are **STRONGLY RECOMMENDED** to have at least 4GB of non-volatile storage for application private data so they will be able to upgrade to the future platform releases.

The Android APIs include a DownloadManager that applications MAY use to download data files. The device implementation of the Download Manager MUST be capable of downloading individual files of at least 100MB in size to the default “cache” location.

7.6.2. Application Shared Storage

Device implementations MUST offer shared storage for applications also often referred as “shared external storage”.

Device implementations MUST be configured with shared storage mounted by default, “out of the box”. If the shared storage is not mounted on the Linuxpath /sdcard, then the device MUST include a Linux symbolic link from /sdcard to the actual mount point.

Device implementations MAY have hardware for user-accessible removable storage, such as a Secure Digital (SD) card slot. If this slot is used to satisfy the shared storage requirement, the device implementation:

- MUST implement a toast or pop-up user interface warning the user when there is no SD card.
- MUST include a FAT-formatted SD card 1GB in size or larger OR show on the box and other material available at time of purchase that the SD card has to be separately purchased.
- MUST mount the SD card by default.

Alternatively, device implementations MAY allocate internal (non-removable) storage as shared storage for apps as included in the upstream Android Open Source Project; device implementations SHOULD use this configuration and software implementation. If a device implementation uses internal (non-removable) storage to satisfy the shared storage requirement, while that storage MAY share space with the application private data, it MUST be at least 1GB in size and mounted on /sdcard (or /sdcard MUST be a symbolic link to the physical location if it is mounted elsewhere).

Device implementations MUST enforce as documented the android.permission.WRITE_EXTERNAL_STORAGE permission on this shared storage. Shared storage MUST otherwise be writable by any application that obtains that permission.

Device implementations that include multiple shared storage paths (such as both an SD card slot and shared internal storage) MUST allow only pre-installed & privileged Android applications with the WRITE_EXTERNAL_STORAGE permission to write to the secondary external storage, except when writing to their package-specific directories or within the URI returned by firing the ACTION_OPEN_DOCUMENT_TREE intent.

However, device implementations SHOULD expose content from both storage paths transparently through Android’s media scanner service and android.provider.MediaStore.

Regardless of the form of shared storage used, if the device implementation has a USB port with USB peripheral mode support, it MUST provide some mechanism to access the contents of shared storage from a host computer. Device implementations MAY use USB mass storage, but SHOULD use Media Transfer Protocol to satisfy this requirement. If the device implementation supports Media Transfer Protocol, it:

- SHOULD be compatible with the reference Android MTP host, Android File Transfer.
- SHOULD report a USB device class of 0x00.
- SHOULD report a USB interface name of 'MTP'.

7.6.3. Adoptable Storage

Device implementations are STRONGLY RECOMMENDED to implement adoptable storage if the removable storage device port is in a long-term stable location, such as within the battery

compartment or other protective cover.

Device implementations such as a television, MAY enable adoption through USB ports as the device is expected to be static and not mobile. But for other device implementations that are mobile in nature, it is STRONGLY RECOMMENDED to implement the adoptable storage in a long-term stable location, since accidentally disconnecting them can cause data loss/corruption.

7.7. USB

Device implementations SHOULD support USB peripheral mode and SHOULD support USB host mode.

7.7.1. USB peripheral mode

If a device implementation includes a USB port supporting peripheral mode:

- The port MUST be connectable to a USB host that has a standard type-A or type-C USB port.
- The port SHOULD use micro-B, micro-AB or Type-C USB form factor. Existing and new Android devices are **STRONGLY RECOMMENDED to meet these requirements** so they will be able to upgrade to the future platform releases.
- The port SHOULD be located on the bottom of the device (according to natural orientation) or enable software screen rotation for all apps (including home screen), so that the display draws correctly when the device is oriented with the port at bottom. Existing and new Android devices are **STRONGLY RECOMMENDED to meet these requirements** so they will be able to upgrade to future platform releases.
- It MUST allow a USB host connected with the Android device to access the contents of the shared storage volume using either USB mass storage or Media Transfer Protocol.
- It SHOULD implement the Android Open Accessory (AOA) API and specification as documented in the Android SDK documentation, and if it is an Android Handheld device it MUST implement the AOA API. Device implementations implementing the AOA specification:
  - MUST declare support for the hardware feature android.hardware.usb.accessory.
  - MUST implement the USB audio class as documented in the Android SDK documentation.
  - The USB mass storage class MUST include the string "android" at the end of the interface description iInterface string of the USB mass storage
- It SHOULD implement support to draw 1.5 A current during HS chirp and traffic as specified in the USB Battery Charging specification, revision 1.2. Existing and new Android devices are **STRONGLY RECOMMENDED to meet these requirements** so they will be able to upgrade to the future platform releases.
- Type-C devices MUST detect 1.5A and 3.0A chargers per the Type-C resistor standard and it must detect changes in the advertisement.
- Type-C devices also supporting USB host mode are STRONGLY RECOMMENDED to support Power Delivery for data and power role swapping.
- Type-C devices SHOULD support Power Delivery for high-voltage charging and support for Alternate Modes such as display out.
- The value of iSerialNumber in USB standard device descriptor MUST be equal to the value of android.os.Build.SERIAL.
- Type-C devices are STRONGLY RECOMMENDED to not support proprietary charging methods that modify Vbus voltage beyond default levels, or alter sink/source roles as such may result in interoperability issues with the chargers or devices that support the standard USB Power Delivery methods. While this is called out as "STRONGLY RECOMMENDED",

in future Android versions we might REQUIRE all type-C devices to support full interoperability with standard type-C chargers.

7.7.2. USB host mode

If a device implementation includes a USB port supporting host mode, it:

- SHOULD use a type-C USB port, if the device implementation supports USB 3.1.
- MAY use a non-standard port form factor, but if so MUST ship with a cable or cables adapting the port to a standard type-A or type-C USB port.
- MAY use a micro-AB USB port, but if so SHOULD ship with a cable or cables adapting the port to a standard type-A or type-C USB port.
- is **STRONGLY RECOMMENDED** to implement the USB audio class as documented in the Android SDK documentation.
- MUST implement the Android USB host API as documented in the Android SDK, and MUST declare support for the hardware feature android.hardware.usb.host.
- SHOULD support the Charging Downstream Port output current range of 1.5 A ~ 5 A as specified in the USB Battery Charging specifications, revision 1.2.
- USB Type-C devices are STRONGLY RECOMMENDED to support DisplayPort, SHOULD support USB SuperSpeed Data Rates, and are STRONGLY RECOMMENDED to support Power Delivery for data and power role swapping.
- Devices with any type-A or type-AB ports MUST NOT ship with an adapter converting from this port to a type-C receptacle.
- MUST recognize any remotely connected MTP (Media Transfer Protocol) devices and make their contents accessible through the ACTION_GET_CONTENT , ACTION_OPEN_DOCUMENT , and ACTION_CREATE_DOCUMENT intents, if the Storage Access Framework (SAF) is supported.
- MUST, if using a Type-C USB port and including support for peripheral mode, implement Dual Role Port functionality as defined by the USB Type-C specification (section 4.5.1.3.3).
- SHOULD, if the Dual Role Port functionality is supported, implement the Try.* model that is most appropriate for the device form factor. For example a handheld device SHOULD implement the Try.SNK model.

7.8. Audio

7.8.1. Microphone

Android Handheld, Watch, and Automotive implementations MUST include a microphone.

Device implementations MAY omit a microphone. However, if a device implementation omits a microphone, it MUST NOT report the android.hardware.microphone feature constant, and MUST implement the audio recording API at least as no-ops, per section 7. Conversely, device implementations that do possess a microphone:

- MUST report the android.hardware.microphone feature constant.
- MUST meet the audio recording requirements in section 5.4.
- MUST meet the audio latency requirements in section 5.6.
- STRONGLY RECOMMENDED to support near-ultrasound recording as described in section 7.8.3.

7.8.2. Audio Output

Android Watch devices MAY include an audio output.

Device implementations including a speaker or with an audio/multimedia output port for an audio output peripheral as a headset or an external speaker:

- MUST report the android.hardware.audio.output feature constant.
- MUST meet the audio playback requirements in section 5.5.
- MUST meet the audio latency requirements in section 5.6.
- STRONGLY RECOMMENDED to support near-ultrasound playback as described in section 7.8.3.

Conversely, if a device implementation does not include a speaker or audio output port, it MUST NOT report the android.hardware.audio output feature, and MUST implement the Audio Output related APIs as no-ops at least.

Android Watch device implementation MAY but SHOULD NOT have audio output, but other types of Android device implementations MUST have an audio output and declare android.hardware.audio.output.

7.8.2.1. Analog Audio Ports

In order to be compatible with the headsets and other audio accessories using the 3.5mm audio plug across the Android ecosystem, if a device implementation includes one or more analog audio ports, at least one of the audio port(s) SHOULD be a 4 conductor 3.5mm audio jack. If a device implementation has a 4 conductor 3.5mm audio jack, it:

- MUST support audio playback to stereo headphones and stereo headsets with a microphone, and SHOULD support audio recording from stereo headsets with a microphone.
- MUST support TRRS audio plugs with the CTIA pin-out order, and SHOULD support audio plugs with the OMTP pin-out order.
- MUST support the detection of microphone on the plugged in audio accessory, if the device implementation supports a microphone, and broadcast the android.intent.action.HEADSET_PLUG with the extra value microphone set as 1.
- MUST support the detection and mapping to the keycodes for the following 3 ranges of equivalent impedance between the microphone and ground conductors on the audio plug:
  - **70 ohm or less** : KEYCODE_HEADSETHOOK
  - **210-290 Ohm** : KEYCODE_VOLUME_UP
  - **360-680 Ohm** : KEYCODE_VOLUME_DOWN
- STRONGLY RECOMMENDED to detect and map to the keycode for the following range of equivalent impedance between the microphone and ground conductors on the audio plug:
  - **110-180 Ohm:** KEYCODE_VOICE_ASSIST
- MUST trigger ACTION_HEADSET_PLUG upon a plug insert, but only after all contacts on plug are touching their relevant segments on the jack.
- MUST be capable of driving at least 150mV ± 10% of output voltage on a 32 Ohm speaker impedance.
- MUST have a microphone bias voltage between 1.8V ~ 2.9V.

7.8.3. Near-Ultrasound

Near-Ultrasound audio is the 18.5 kHz to 20 kHz band. Device implementations MUST correctly report the support of near-ultrasound audio capability via the AudioManager.getProperty API as follows:

- If PROPERTY_SUPPORT_MIC_NEAR_ULTRASOUND is "true", then the following requirements must be met by the VOICE_RECOGNITION and UNPROCESSED audio

sources:
    - The microphone's mean power response in the 18.5 kHz to 20 kHz band MUST be no more than 15 dB below the response at 2 kHz.
    - The microphone's unweighted signal to noise ratio over 18.5 kHz to 20 kHz for a 19 kHz tone at -26 dBFS MUST be no lower than 50 dB.
- If PROPERTY_SUPPORT_SPEAKER_NEAR_ULTRASOUND is "true", then the speaker's mean response in 18.5 kHz - 20 kHz MUST be no lower than 40 dB below the response at 2 kHz.

7.9. Virtual Reality

Android includes APIs and facilities to build "Virtual Reality" (VR) applications including high quality mobile VR experiences. Device implementations MUST properly implement these APIs and behaviors, as detailed in this section.

7.9.1. Virtual Reality Mode

Android handheld device implementations that support a mode for VR applications that handles stereoscopic rendering of notifications and disable monocular system UI components while a VR application has user focus MUST declare android.software.vr.mode feature. Devices declaring this feature MUST include an application implementing android.service.vr.VrListenerService that can be enabled by VR applications via android.app.Activity#setVrModeEnabled .

7.9.2. Virtual Reality High Performance

Android handheld device implementations MUST identify the support of high performance virtual reality for longer user periods through the android.hardware.vr.high_performance feature flag and meet the following requirements.

- Device implementations MUST have at least 2 physical cores.
- Device implementations MUST declare android.software.vr.mode feature.
- Device implementations MUST provide an exclusive core to the foreground application and MUST support the Process.getExclusiveCores API to return the numbers of the cpu cores that are exclusive to the top foreground application. This core MUST not allow any other userspace processes to run on it (except device drivers used by the application), but MAY allow some kernel processes to run as necessary.
- Device implementations MUST support sustained performance mode.
- Device implementations MUST support OpenGL ES 3.2.
- Device implementations MUST support Vulkan Hardware Level 0 and SHOULD support Vulkan Hardware Level 1.
- Device implementations MUST implement EGL_KHR_mutable_render_buffer and EGL_ANDROID_front_buffer_auto_refresh, EGL_ANDROID_create_native_client_buffer, EGL_KHR_fence_sync and EGL_KHR_wait_sync so that they may be used for Shared Buffer Mode, and expose the extensions in the list of available EGL extensions.
- The GPU and display MUST be able to synchronize access to the shared front buffer such that alternating-eye rendering of VR content at 60fps with two render contexts will be displayed with no visible tearing artifacts.
- Device implementations MUST implement EGL_IMG_context_priority, and expose the extension in the list of available EGL extensions.
- Device implementations MUST implement GL_EXT_multisampled_render_to_texture, GL_OVR_multiview, GL_OVR_multiview2 and GL_OVR_multiview_multisampled_render_to_texture, and expose the extensions in the list of available GL extensions.

- Device implementations MUST implement EGL_EXT_protected_content and GL_EXT_protected_textures so that it may be used for Secure Texture Video Playback, and expose the extensions in the list of available EGL and GL extensions.
- Device implementations MUST support H.264 decoding at least 3840x2160@30fps-40Mbps (equivalent to 4 instances of 1920x1080@30fps-10Mbps or 2 instances of 1920x1080@60fps-20Mbps).
- Device implementations MUST support HEVC and VP9, MUST be capable to decode at least 1920x1080@30fps-10Mbps and SHOULD be capable to decode 3840x2160@30fps-20Mbps (equivalent to 4 instances of 1920x1080@30fps-5Mbps).
- The device implementations are STRONGLY RECOMMENDED to support android.hardware.sensor.hifi_sensors feature and MUST meet the gyroscope, accelerometer, and magnetometer related requirements for android.hardware.hifi_sensors.
- Device implementations MUST support HardwarePropertiesManager.getDeviceTemperatures API and return accurate values for skin temperature.
- The device implementation MUST have an embedded screen, and its resolution MUST be at least be FullHD(1080p) and STRONGLY RECOMMENDED TO BE be QuadHD (1440p) or higher.
- The display MUST measure between 4.7" and 6" diagonal.
- The display MUST update at least 60 Hz while in VR Mode.
- The display latency on Gray-to-Gray, White-to-Black, and Black-to-White switching time MUST be ≤ 3 ms.
- The display MUST support a low-persistence mode with ≤5 ms persistence,persistence being defined as the amount of time for which a pixel is emitting light.
- Device implementations MUST support Bluetooth 4.2 and Bluetooth LE Data Length Extension section 7.4.3.

8. Performance and Power

Some minimum performance and power criteria are critical to the user experience and impact the baseline assumptions developers would have when developing an app. Android Watch devices SHOULD and other type of device implementations MUST meet the following criteria.

8.1. User Experience Consistency

Device implementations MUST provide a smooth user interface by ensuring a consistent frame rate and response times for applications and games. Device implementations MUST meet the following requirements:

- **Consistent frame latency** . Inconsistent frame latency or a delay to render frames MUST NOT happen more often than 5 frames in a second, and SHOULD be below 1 frames in a second.
- **User interface latency** . Device implementations MUST ensure low latency user experience by scrolling a list of 10K list entries as defined by the Android Compatibility Test Suite (CTS) in less than 36 secs.
- **Task switching** . When multiple applications have been launched, re-launching an already-running application after it has been launched MUST take less than 1 second.

8.2. File I/O Access Performance

Device implementations MUST ensure internal storage file access performance consistency for read and write operations.

- **Sequential write** . Device implementations MUST ensure a sequential write performance of at least 5MB/s for a 256MB file using 10MB write buffer.
- **Random write** . Device implementations MUST ensure a random write performance of at least 0.5MB/s for a 256MB file using 4KB write buffer.
- **Sequential read** . Device implementations MUST ensure a sequential read performance of at least 15MB/s for a 256MB file using 10MB write buffer.
- **Random read** . Device implementations MUST ensure a random read performance of at least 3.5MB/s for a 256MB file using 4KB write buffer.

8.3. Power-Saving Modes

Android 6.0 introduced App Standby and Doze power-saving modes to optimize battery usage. All Apps exempted from these modes MUST be made visible to the end user. Further, the triggering, maintenance, wakeup algorithms and the use of global system settings of these power-saving modes MUST not deviate from the Android Open Source Project.

In addition to the power-saving modes, Android device implementations MAY implement any or all of the 4 sleeping power states as defined by the Advanced Configuration and Power Interface (ACPI), but if it implements S3 and S4 power states, it can only enter these states when closing a lid that is physically part of the device.

8.4. Power Consumption Accounting

A more accurate accounting and reporting of the power consumption provides the app developer both the incentives and the tools to optimize the power usage pattern of the application. Therefore, device implementations:

- MUST be able to track hardware component power usage and attribute that power usage to specific applications. Specifically, implementations:
  - MUST provide a per-component power profile that defines the current consumption value for each hardware component and the approximate battery drain caused by the components over time as documented in the Android Open Source Project site.
  - MUST report all power consumption values in milliampere hours (mAh).
  - SHOULD be attributed to the hardware component itself if unable to attribute hardware component power usage to an application.
  - MUST report CPU power consumption per each process's UID. The Android Open Source Project meets the requirement through the uid_cputime kernel module implementation.
- MUST make this power usage available via the adb shell dumpsys batterystats shell command to the app developer.
- MUST honor the android.intent.action.POWER_USAGE_SUMMARY intent and display a settings menu that shows this power usage.

8.5. Consistent Performance

Performance can fluctuate dramatically for high-performance long-running apps, either because of the other apps running in the background or the CPU throttling due to temperature limits. Android includes programmatic interfaces so that when the device is capable, the top foreground application can request that the system optimize the allocation of the resources to address such fluctuations.

Device implementations SHOULD support Sustained Performance Mode which can provide the top foreground application a consistent level of performance for a prolonged amount of time when requested through the Window.setSustainedPerformanceMode() API method. A Device

implementation MUST report the support of Sustained Performance Mode accurately through the PowerManager.isSustainedPerformanceModeSupported() API method.

Device implementations with two or more CPU cores SHOULD provide at least one exclusive core that can be reserved by the top foreground application. If provided, implementations MUST meet the following requirements:

- Implementations MUST report through the Process.getExclusiveCores() API method the id numbers of the exclusive cores that can be reserved by the top foreground application.
- Device implementations MUST not allow any user space processes except the device drivers used by the application to run on the exclusive cores, but MAY allow some kernel processes to run as necessary.

If a device implementation does not support an exclusive core, it MUST return an empty list through the Process.getExclusiveCores() API method.

9. Security Model Compatibility

Device implementations MUST implement a security model consistent with the Android platform security model as defined in Security and Permissions reference document in the APIs in the Android developer documentation. Device implementations MUST support installation of self-signed applications without requiring any additional permissions/certificates from any third parties/authorities. Specifically, compatible devices MUST support the security mechanisms described in the follow subsections.

9.1. Permissions

Device implementations MUST support the Android permissions model as defined in the Android developer documentation. Specifically, implementations MUST enforce each permission defined as described in the SDK documentation; no permissions may be omitted, altered, or ignored. Implementations MAY add additional permissions, provided the new permission ID strings are not in the android.* namespace.

Permissions with a protectionLevel of 'PROTECTION_FLAG_PRIVILEGED' MUST only be granted to apps preloaded in the whitelisted privileged path(s) of the system image, such as the system/priv-app path in the AOSP implementation.

Permissions with a protection level of dangerous are runtime permissions. Applications with targetSdkVersion > 22 request them at runtime. Device implementations:

- MUST show a dedicated interface for the user to decide whether to grant the requested runtime permissions and also provide an interface for the user to manage runtime permissions.
- MUST have one and only one implementation of both user interfaces.
- MUST NOT grant any runtime permissions to preinstalled apps unless:
  - the user's consent can be obtained before the application uses it
  - the runtime permissions are associated with an intent pattern for which the preinstalled application is set as the default handler

9.2. UID and Process Isolation

Device implementations MUST support the Android application sandbox model, in which each application runs as a unique Unixstyle UID and in a separate process. Device implementations MUST support running multiple applications as the same Linux user ID, provided that the applications are properly signed and constructed, as defined in the Security and Permissions reference.

9.3. Filesystem Permissions

Device implementations MUST support the Android file access permissions model as defined in the Security and Permissions reference.

9.4. Alternate Execution Environments

Device implementations MAY include runtime environments that execute applications using some other software or technology than the Dalvik Executable Format or native code. However, such alternate execution environments MUST NOT compromise the Android security model or the security of installed Android applications, as described in this section.

Alternate runtimes MUST themselves be Android applications, and abide by the standard Android security model, as described elsewhere in section 9.

Alternate runtimes MUST NOT be granted access to resources protected by permissions not requested in the runtime's AndroidManifest.xml file via the <uses-permission> mechanism.

Alternate runtimes MUST NOT permit applications to make use of features protected by Android permissions restricted to system applications.

Alternate runtimes MUST abide by the Android sandbox model. Specifically, alternate runtimes:

- SHOULD install apps via the PackageManager into separate Android sandboxes (Linux user IDs, etc.).
- MAY provide a single Android sandbox shared by all applications using the alternate runtime.
- Installed applications using an alternate runtime MUST NOT reuse the sandbox of any other app installed on the device, except through the standard Android mechanisms of shared user ID and signing certificate.
- MUST NOT launch with, grant, or be granted access to the sandboxes corresponding to other Android applications.
- MUST NOT be launched with, be granted, or grant to other applications any privileges of the superuser (root), or of any other user ID.

The .apk files of alternate runtimes MAY be included in the system image of a device implementation, but MUST be signed with a key distinct from the key used to sign other applications included with the device implementation.

When installing applications, alternate runtimes MUST obtain user consent for the Android permissions used by the application. If an application needs to make use of a device resource for which there is a corresponding Android permission (such as Camera, GPS, etc.), the alternate runtime MUST inform the user that the application will be able to access that resource. If the runtime environment does not record application capabilities in this manner, the runtime environment MUST list all permissions held by the runtime itself when installing any application using that runtime.

9.5. Multi-User Support

This feature is optional for all device types.

Android includes support for multiple users and provides support for full user isolation. Device implementations MAY enable multiple users, but when enabled MUST meet the following requirements related to multi-user support:

- Android Automotive device implementations with multi-user support enabled MUST include a guest account that allows all functions provided by the vehicle system without requiring a user to log in.
- Device implementations that do not declare the android.hardware.telephony feature flag

MUST support restricted profiles, a feature that allows device owners to manage additional users and their capabilities on the device. With restricted profiles, device owners can quickly set up separate environments for additional users to work in, with the ability to manage finer-grained restrictions in the apps that are available in those environments.

- Conversely device implementations that declare the android.hardware.telephony feature flag MUST NOT support restricted profiles but MUST align with the AOSP implementation of controls to enable /disable other users from accessing the voice calls and SMS.
- Device implementations MUST, for each user, implement a security model consistent with the Android platform security model as defined in Security and Permissions reference document in the APIs.
- Each user instance on an Android device MUST have separate and isolated external storage directories. Device implementations MAY store multiple users' data on the same volume or filesystem. However, the device implementation MUST ensure that applications owned by and running on behalf a given user cannot list, read, or write to data owned by any other user. Note that removable media, such as SD card slots, can allow one user to access another's data by means of a host PC. For this reason, device implementations that use removable media for the external storage APIs MUST encrypt the contents of the SD card if multiuser is enabled using a key stored only on non-removable media accessible only to the system. As this will make the media unreadable by a host PC, device implementations will be required to switch to MTP or a similar system to provide host PCs with access to the current user's data. Accordingly, device implementations MAY but SHOULD NOT enable multi-user if they use removable media for primary external storage.

9.6. Premium SMS Warning

Android includes support for warning users of any outgoing premium SMS message. Premium SMS messages are text messages sent to a service registered with a carrier that may incur a charge to the user. Device implementations that declare support for android.hardware.telephony MUST warn users before sending a SMS message to numbers identified by regular expressions defined in /data/misc/sms/codes.xml file in the device. The upstream Android Open Source Project provides an implementation that satisfies this requirement.

9.7. Kernel Security Features

The Android Sandbox includes features that use the Security-Enhanced Linux (SELinux) mandatory access control (MAC) system, seccomp sandboxing, and other security features in the Linux kernel. SELinux or any other security features implemented below the Android framework:

- MUST maintain compatibility with existing applications.
- MUST NOT have a visible user interface when a security violation is detected and successfully blocked, but MAY have a visible user interface when an unblocked security violation occurs resulting in a successful exploit.
- SHOULD NOT be user or developer configurable.

If any API for configuration of policy is exposed to an application that can affect another application (such as a Device Administration API), the API MUST NOT allow configurations that break compatibility.

Devices MUST implement SELinux or, if using a kernel other than Linux, an equivalent mandatory access control system. Devices MUST also meet the following requirements, which are satisfied by the reference implementation in the upstream Android Open Source Project.

Device implementations:

- MUST set SELinux to global enforcing mode.
- MUST configure all domains in enforcing mode. No permissive mode domains are allowed, including domains specific to a device/vendor.
- MUST NOT modify, omit, or replace the neverallow rules present within the system/sepolicy folder provided in the upstream Android Open Source Project (AOSP) and the policy MUST compile with all neverallow rules present, for both AOSP SELinux domains as well as device/vendor specific domains.
- MUST split the media framework into multiple processes so that it is possible to more narrowly grant access for each process as described in the Android Open Source Project site.

Device implementations SHOULD retain the default SELinux policy provided in the system/sepolicy folder of the upstream Android Open Source Project and only further add to this policy for their own device-specific configuration. Device implementations MUST be compatible with the upstream Android Open Source Project.

Devices MUST implement a kernel application sandboxing mechanism which allows filtering of system calls using a configurable policy from multithreaded programs. The upstream Android Open Source Project meets this requirement through enabling the seccomp-BPF with threadgroup synchronization (TSYNC) as described in the Kernel Configuration section of source.android.com.

9.8. Privacy

If the device implements functionality in the system that captures the contents displayed on the screen and/or records the audio stream played on the device, it MUST continuously notify the user whenever this functionality is enabled and actively capturing/recording.

If a device implementation has a mechanism that routes network data traffic through a proxy server or VPN gateway by default (for example, preloading a VPN service with android.permission.CONTROL_VPN granted), the device implementation MUST ask for the user's consent before enabling that mechanism, unless that VPN is enabled by the Device Policy Controller via the DevicePolicyManager.setAlwaysOnVpnPackage() , in which case the user does not need to provide a separate consent, but MUST only be notified.

Device implementations MUST ship with an empty user-added Certificate Authority (CA) store, and MUST preinstall the same root certificates for the system-trusted CA store as provided in the upstream Android Open Source Project.

When devices are routed through a VPN, or a user root CA is installed, the implementation MUST display a warning indicating the network traffic may be monitored to the user.

If a device implementation has a USB port with USB peripheral mode support, it MUST present a user interface asking for the user's consent before allowing access to the contents of the shared storage over the USB port.

9.9. Data Storage Encryption

Optional for Android device implementations without a secure lock screen.

If the device implementation supports a secure lock screen as described in section 9.11.1, then the device MUST support data storage encryption of the application private data (/data partition), as well as the application shared storage partition (/sdcard partition) if it is a permanent, non-removable part of the device.

For device implementations supporting data storage encryption and with Advanced Encryption Standard (AES) crypto performance above 50MiB/sec, the data storage encryption MUST be enabled by default at the time the user has completed the out-of-box setup experience. If a device implementation is already launched on an earlier Android version with encryption disabled by default, such a device cannot meet the requirement through a system software update and thus MAY be

exempted.

Device implementations SHOULD meet the above data storage encryption requirement via implementing File Based Encryption (FBE).

9.9.1. Direct Boot

All devices MUST implement the Direct Boot mode APIs even if they do not support Storage Encryption. In particular, the LOCKED_BOOT_COMPLETED(https://developer.android.com/reference/android/content/Intent.html#LOCK and ACTION_USER_UNLOCKED Intents must still be broadcast to signal Direct Boot aware applications that Device Encrypted (DE) and Credential Encrypted (CE) storage locations are available for user.

9.9.2. File Based Encryption

Device implementations supporting FBE:

- MUST boot up without challenging the user for credentials and allow Direct Boot aware apps to access to the Device Encrypted (DE) storage after the LOCKED_BOOT_COMPLETED message is broadcasted.
- MUST only allow access to Credential Encrypted (CE) storage after the user has unlocked the device by supplying their credentials (eg. passcode, pin, pattern or fingerprint) and the ACTION_USER_UNLOCKED message is broadcasted. Device implementations MUST NOT offer any method to unlock the CE protected storage without the user supplied credentials.
- MUST support Verified Boot and ensure that DE keys are cryptographically bound to the device's hardware root of trust.
- MUST support encrypting file contents using AES with a key length of 256-bits in XTS mode.
- MUST support encrypting file name using AES with a key length of 256-bits in CBC-CTS mode.
- MAY support alternative ciphers, key lengths and modes for file content and file name encryption, but MUST use the mandatorily supported ciphers, key lengths and modes by default.
- SHOULD make preloaded essential apps (e.g. Alarm, Phone, Messenger) Direct Boot aware.

The keys protecting CE and DE storage areas:

- MUST be cryptographically bound to a hardware-backed Keystore. CE keys must be bound to a user's lock screen credentials. If the user has specified no lock screen credentials then the CE keys MUST be bound to a default passcode.
- MUST be unique and distinct, in other words no user's CE or DE key may match any other user's CE or DE keys.

The upstream Android Open Source project provides a preferred implementation of this feature based on the Linux kernel ext4 encryption feature.

9.9.3. Full Disk Encryption

Device implementations supporting full disk encryption (FDE). MUST use AES with a key of 128-bits (or greater) and a mode designed for storage (for example, AES-XTS, AES-CBC-ESSIV). The encryption key MUST NOT be written to storage at any time without being encrypted. The user MUST

be provided with the possibility to AES encrypt the encryption key, except when it is in active use, with the lock screen credentials stretched using a slow stretching algorithm (e.g. PBKDF2 or scrypt). If the user has not specified a lock screen credentials or has disabled use of the passcode for encryption, the system SHOULD use a default passcode to wrap the encryption key. If the device provides a hardware-backed keystore, the password stretching algorithm MUST be cryptographically bound to that keystore. The encryption key MUST NOT be sent off the device (even when wrapped with the user passcode and/or hardware bound key). The upstream Android Open Source project provides a preferred implementation of this feature based on the Linux kernel feature dm-crypt.

9.10. Device Integrity

The following requirements ensures there is transparancy to the status of the device integrity.

Device implementations MUST correctly report through the System API method PersistentDataBlockManager.getFlashLockState() whether their bootloader state permits flashing of the system image. The FLASH_LOCK_UNKNOWN state is reserved for device implementations upgrading from an earlier version of Android where this new system API method did not exist.

Verified boot is a feature that guarantees the integrity of the device software. If a device implementation supports the feature, it MUST:

- Declare the platform feature flag android.software.verified_boot.
- Perform verification on every boot sequence.
- Start verification from an immutable hardware key that is the root of trust and go all the way up to the system partition.
- Implement each stage of verification to check the integrity and authenticity of all the bytes in the next stage before executing the code in the next stage.
- Use verification algorithms as strong as current recommendations from NIST for hashing algorithms (SHA-256) and public key sizes (RSA-2048).
- MUST NOT allow boot to complete when system verification fails, unless the user consents to attempt booting anyway, in which case the data from any non-verified storage blocks MUST not be used.
- MUST NOT allow verified partitions on the device to be modified unless the user has explicitly unlocked the boot loader.

The upstream Android Open Source Project provides a preferred implementation of this feature based on the Linux kernel feature dm-verity.

Starting from Android 6.0, device implementations with Advanced Encryption Standard (AES) crypto performance above 50 MiB/seconds MUST support verified boot for device integrity.

If a device implementation is already launched without supporting verified boot on an earlier version of Android, such a device can not add support for this feature with a system software update and thus are exempted from the requirement.

9.11. Keys and Credentials

The Android Keystore System allows app developers to store cryptographic keys in a container and use them in cryptographic operations through the KeyChain API or the Keystore API.

All Android device implementations MUST meet the following requirements:

- SHOULD not limit the number of keys that can be generated, and MUST at least allow more than 8,192 keys to be imported.
- The lock screen authentication MUST rate limit attempts and MUST have an exponential backoff algorithm. Beyond 150 failed attempts, the delay MUST be at least 24 hours per attempt.

- When the device implementation supports a secure lock screen it MUST back up the keystore implementation with secure hardware and meet following requirements:
  - MUST have hardware backed implementations of RSA, AES, ECDSA and HMAC cryptographic algorithms and MD5, SHA1, SHA-2 Family hash functions to properly support the Android Keystore system's supported algorithms.
  - MUST perform the lock screen authentication in the secure hardware and only when successful allow the authentication-bound keys to be used. The upstream Android Open Source Project provides the Gatekeeper Hardware Abstraction Layer (HAL) that can be used to satisfy this requirement.

Note that if a device implementation is already launched on an earlier Android version, and does not have a fingerprint scanner, such a device is exempted from the requirement to have a hardware-backed keystore.

9.11.1. Secure Lock Screen

Device implementations MAY add or modify the authentication methods to unlock the lock screen, but MUST still meet the following requirements:

- The authentication method, if based on a known secret, MUST NOT be treated as a secure lock screen unless it meets all following requirements:
  - The entropy of the shortest allowed length of inputs MUST be greater than 10 bits.
  - The maximum entropy of all possible inputs MUST be greater than 18 bits.
  - MUST not replace any of the existing authentication methods (PIN, pattern, password) implemented and provided in AOSP.
  - MUST be disabled when the Device Policy Controller (DPC) application has set the password quality policy via the DevicePolicyManager.setPasswordQuality() method with a more restrictive quality constant than PASSWORD_QUALITY_SOMETHING .
- The authenticaion method, if based on a physical token or the location, MUST NOT be treated as a secure lock screen unless it meets all following requirements:
  - It MUST have a fall-back mechanism to use one of the primary authentication methods which is based on a known secret and meets the requirements to be treated as a secure lock screen.
  - It MUST be disabled and only allow the primary authentication to unlock the screen when the Device Policy Controller (DPC) application has set the policy with either the DevicePolicyManager.setKeyguardDisabledFeatures(KEYGUARD_DISABLE_TRUST_AGENTS) method or the DevicePolicyManager.setPasswordQuality() method with a more restrictive quality constant than PASSWORD_QUALITY_UNSPECIFIED .
- The authentication method, if based on biometrics, MUST NOT be treated as a secure lock screen unless it meets all following requirements:
  - It MUST have a fall-back mechanism to use one of the primary authentication methods which is based on a known secret and meets the requirements to be treated as a secure lock screen.
  - It MUST be disabled and only allow the primary authentication to unlock the screen when the Device Policy Controller (DPC) application has set the keguard feature policy by calling the method DevicePolicyManager.setKeyguardDisabledFeatures(KEYGUARD_DISABLE_FINGERPRINT) .
  - It MUST have a false acceptance rate that is equal or stronger than what is required for a fingerprint sensor as described in section 7.3.10, or otherwise

MUST be disabled and only allow the primary authentication to unlock the screen when the Device Policy Controller (DPC) application has set the password quality policy via the DevicePolicyManager.setPasswordQuality() method with a more restrictive quality constant than PASSWORD_QUALITY_BIOMETRIC_WEAK .

- If the authentication method can not be treated as a secure lock screen, it:
  - MUST return false for both the KeyguardManager.isKeyguardSecure() and the KeyguardManager.isDeviceSecure() methods.
  - MUST be disabled when the Device Policy Controller (DPC) application has set the password quality policy via the DevicePolicyManager.setPasswordQuality() method with a more restrictive quality constant than PASSWORD_QUALITY_UNSPECIFIED .
  - MUST NOT reset the password expiration timers set by DevicePolicyManager.setPasswordExpirationTimeout() .
  - MUST NOT authenticate access to keystores if the application has called\ KeyGenParameterSpec.Builder.setUserAuthenticationRequired(true) .
- If the authentication method is based on a physical token, the location, or biometrics that has higher false acceptance rate than what is required for fingerprint sensors as described in section 7.3.10, then it:
  - MUST NOT reset the password expiration timers set by DevicePolicyManager.setPasswordExpirationTimeout() .
  - MUST NOT authenticate access to keystores if the application has called KeyGenParameterSpec.Builder.setUserAuthenticationRequired(true) .

9.12. Data Deletion

Devices MUST provide users with a mechanism to perform a "Factory Data Reset" that allows logical and physical deletion of all data except for the following:

- The system image
- Any operating system files required by the system image

All user-generated data MUST be deleted. This MUST satisfy relevant industry standards for data deletion such as NIST SP800-88. This MUST be used for the implementation of the wipeData() API (part of the Android Device Administration API) described in section 3.9 Device Administration.

Devices MAY provide a fast data wipe that conducts a logical data erase.

9.13. Safe Boot Mode

Android provides a mode enabling users to boot up into a mode where only preinstalled system apps are allowed to run and all third-party apps are disabled. This mode, known as "Safe Boot Mode", provides the user the capability to uninstall potentially harmful third-party apps.

Android device implementations are STRONGLY RECOMENDED to implement Safe Boot Mode and meet following requirements:

- Device implementations SHOULD provide the user an option to enter Safe Boot Mode from the boot menu which is reachable through a workflow that is different from that of normal boot.
- Device implementations MUST provide the user an option to enter Safe Boot Mode in such a way that is uninterruptible from third-party apps installed on the device, except for when the third party app is a Device Policy Controller and has set the UserManager.DISALLOW_SAFE_BOOT flag as true.

- Device implementations MUST provide the user the capability to uninstall any third-party apps within Safe Mode.

9.14. Automotive Vehicle System Isolation

Android Automotive devices are expected to exchange data with critical vehicle subsystems, e.g., by using the vehicle HAL to send and receive messages over vehicle networks such as CAN bus. Android Automotive device implementations MUST implement security features below the Android framework layers to prevent malicious or unintentional interaction between the Android framework or third-party apps and vehicle subsystems. These security features are as follows:

- Gatekeeping messages from Android framework vehicle subsystems, e.g., whitelisting permitted message types and message sources.
- Watchdog against denial of service attacks from the Android framework or third-party apps. This guards against malicious software flooding the vehicle network with traffic, which may lead to malfunctioning vehicle subsystems.

10. Software Compatibility Testing

Device implementations MUST pass all tests described in this section.

However, note that no software test package is fully comprehensive. For this reason, device implementers are **STRONGLY RECOMMENDED** to make the minimum number of changes as possible to the reference and preferred implementation of Android available from the Android Open Source Project. This will minimize the risk of introducing bugs that create incompatibilities requiring rework and potential device updates.

10.1. Compatibility Test Suite

Device implementations MUST pass the Android Compatibility Test Suite (CTS) available from the Android Open Source Project, using the final shipping software on the device. Additionally, device implementers SHOULD use the reference implementation in the Android Open Source tree as much as possible, and MUST ensure compatibility in cases of ambiguity in CTS and for any reimplementations of parts of the reference source code.

The CTS is designed to be run on an actual device. Like any software, the CTS may itself contain bugs. The CTS will be versioned independently of this Compatibility Definition, and multiple revisions of the CTS may be released for Android 7.1. Device implementations MUST pass the latest CTS version available at the time the device software is completed.

10.2. CTS Verifier

Device implementations MUST correctly execute all applicable cases in the CTS Verifier. The CTS Verifier is included with the Compatibility Test Suite, and is intended to be run by a human operator to test functionality that cannot be tested by an automated system, such as correct functioning of a camera and sensors.

The CTS Verifier has tests for many kinds of hardware, including some hardware that is optional. Device implementations MUST pass all tests for hardware that they possess; for instance, if a device possesses an accelerometer, it MUST correctly execute the Accelerometer test case in the CTS Verifier. Test cases for features noted as optional by this Compatibility Definition Document MAY be skipped or omitted.

Every device and every build MUST correctly run the CTS Verifier, as noted above. However, since many builds are very similar, device implementers are not expected to explicitly run the CTS Verifier on builds that differ only in trivial ways. Specifically, device implementations that differ from an

implementation that has passed the CTS Verifier only by the set of included locales, branding, etc. MAY omit the CTS Verifier test.

11. Updatable Software

Device implementations MUST include a mechanism to replace the entirety of the system software. The mechanism need not perform “live” upgrades—that is, a device restart MAY be required.

Any method can be used, provided that it can replace the entirety of the software preinstalled on the device. For instance, any of the following approaches will satisfy this requirement:

- “Over-the-air (OTA)” downloads with offline update via reboot.
- “Tethered” updates over USB from a host PC.
- “Offline” updates via a reboot and update from a file on removable storage.

However, if the device implementation includes support for an unmetered data connection such as 802.11 or Bluetooth PAN (Personal Area Network) profile, it MUST support OTA downloads with offline update via reboot.

The update mechanism used MUST support updates without wiping user data. That is, the update mechanism MUST preserve application private data and application shared data. Note that the upstream Android software includes an update mechanism that satisfies this requirement.

For device implementations that are launching with Android 6.0 and later, the update mechanism SHOULD support verifying that the system image is binary identical to expected result following an OTA. The block-based OTA implementation in the upstream Android Open Source Project, added since Android 5.1, satisfies this requirement.

Also, device implementations SHOULD support A/B system updates. The AOSP implements this feature using the boot control HAL.

If an error is found in a device implementation after it has been released but within its reasonable product lifetime that is determined in consultation with the Android Compatibility Team to affect the compatibility of third-party applications, the device implementer MUST correct the error via a software update available that can be applied per the mechanism just described.

Android includes features that allow the Device Owner app (if present) to control the installation of system updates. To facilitate this, the system update subsystem for devices that report android.software.device_admin MUST implement the behavior described in the SystemUpdatePolicy class.

12. Document Changelog

For a summary of changes to the Compatibility Definition in this release:

- Document changelog

For a summary of changes to individuals sections:

1. Introduction
2. Device Types
3. Software
4. Application Packaging
5. Multimedia
6. Developer Tools and Options
7. Hardware Compatibility
8. Performance and Power

12.1. Changelog Viewing Tips

Changes are marked as follows:

- **CDD**
  Substantive changes to the compatibility requirements.
- **Docs**
  Cosmetic or build related changes.

For best viewing, append the pretty=full and no-merges URL parameters to your changelog URLs.

13. Contact Us

You can join the android-compatibility forum and ask for clarifications or bring up any issues that you think the document does not cover.