android compatibility program

Android 4.2 Compatibility Definition

Revision 3

Last updated: June 10, 2013

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 4.2.

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 [Resources, 1].

As used in this document, a "device implementer" or "implementer" is a person or organization developing a hardware/software solution running Android 4.2. A "device implementation" or "implementation" is the hardware/software solution so developed.

To be considered compatible with Android 4.2, 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 [Resources, 3] is both the reference and preferred implementation of Android. Device implementers are strongly encouraged 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 this practice is strongly discouraged, 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.

2. Resources

1. IETF RFC2119 Requirement Levels: http://www.ietf.org/rfc/rfc2119.txt
2. Android Compatibility Program Overview: http://source.android.com/compatibility/index.html
3. Android Open Source Project: http://source.android.com/
4. API definitions and documentation: http://developer.android.com/reference/packages.html
5. Android Permissions reference: http://developer.android.com/reference/android/Manifest.permission.html
6. android.os.Build reference: http://developer.android.com/reference/android/os/Build.html
7. Android 4.2 allowed version strings: http://source.android.com/compatibility/4.2/versions.html
8. Renderscript: http://developer.android.com/guide/topics/graphics/renderscript.html
9. Hardware Acceleration: http://developer.android.com/guide/topics/graphics/hardware-accel.html
10. android.webkit.WebView class: http://developer.android.com/reference/android/webkit/WebView.html
11. HTML5: http://www.whatwg.org/specs/web-apps/current-work/multipage/
12. HTML5 offline capabilities: http://dev.w3.org/html5/spec/Overview.html#offline
13. HTML5 video tag: http://dev.w3.org/html5/spec/Overview.html#video
14. HTML5/W3C geolocation API: http://www.w3.org/TR/geolocation-API/
15. HTML5/W3C webdatabase API: http://www.w3.org/TR/webdatabase/
16. HTML5/W3C IndexedDB API: http://www.w3.org/TR/IndexedDB/
17. Dalvik Virtual Machine specification: available in the Android source code, at dalvik/docs
18. AppWidgets: http://developer.android.com/guide/practices/ui_guidelines/widget_design.html
19. Notifications: http://developer.android.com/guide/topics/ui/notifiers/notifications.html
20. Application Resources: http://code.google.com/android/reference/available-resources.html
21. Status Bar icon style guide: http://developer.android.com/guide/practices/ui_guidelines/icon_design_status_bar.html
22. Search Manager: http://developer.android.com/reference/android/app/SearchManager.html
23. Toasts: http://developer.android.com/reference/android/widget/Toast.html
24. Themes: http://developer.android.com/guide/topics/ui/themes.html

25. R.style class: http://developer.android.com/reference/android/R.style.html
26. Live Wallpapers: http://developer.android.com/resources/articles/live-wallpapers.html
27. Android Device Administration: http://developer.android.com/guide/topics/admin/device-admin.html
28. DevicePolicyManager reference: http://developer.android.com/reference/android/app/admin/DevicePolicyManager.html
29. Android Accessibility Service APIs: http://developer.android.com/reference/android/accessibilityservice/package-summary.html
30. Android Accessibility APIs: http://developer.android.com/reference/android/view/accessibility/package-summary.html
31. Eyes Free project: http://code.google.com/p/eyes-free
32. Text-To-Speech APIs: http://developer.android.com/reference/android/speech/tts/package-summary.html
33. Reference tool documentation (for adb, aapt, ddms, systrace): http://developer.android.com/guide/developing/tools/index.html
34. Android apk file description: http://developer.android.com/guide/topics/fundamentals.html
35. Manifest files: http://developer.android.com/guide/topics/manifest/manifest-intro.html
36. Monkey testing tool: http://developer.android.com/guide/developing/tools/monkey.html
37. Android android.content.pm.PackageManager class and Hardware Features List: http://developer.android.com/reference/android/content/pm/PackageManager.html
38. Supporting Multiple Screens: http://developer.android.com/guide/practices/screens_support.html
39. android.util.DisplayMetrics: http://developer.android.com/reference/android/util/DisplayMetrics.html
40. android.content.res.Configuration: http://developer.android.com/reference/android/content/res/Configuration.html
41. android.hardware.SensorEvent: http://developer.android.com/reference/android/hardware/SensorEvent.html
42. Bluetooth API: http://developer.android.com/reference/android/bluetooth/package-summary.html
43. NDEF Push Protocol: http://source.android.com/compatibility/ndef-push-protocol.pdf
44. MIFARE MF1S503X: http://www.nxp.com/documents/data_sheet/MF1S503x.pdf
45. MIFARE MF1S703X: http://www.nxp.com/documents/data_sheet/MF1S703x.pdf
46. MIFARE MF0ICU1: http://www.nxp.com/documents/data_sheet/MF0ICU1.pdf
47. MIFARE MF0ICU2: http://www.nxp.com/documents/short_data_sheet/MF0ICU2_SDS.pdf
48. MIFARE AN130511: http://www.nxp.com/documents/application_note/AN130511.pdf
49. MIFARE AN130411: http://www.nxp.com/documents/application_note/AN130411.pdf
50. Camera orientation API: http://developer.android.com/reference/android/hardware/Camera.html#setDisplayOrientation(int)
51. Camera: http://developer.android.com/reference/android/hardware/Camera.html
52. Android Open Accessories: http://developer.android.com/guide/topics/usb/accessory.html
53. USB Host API: http://developer.android.com/guide/topics/usb/host.html
54. Android Security and Permissions reference: http://developer.android.com/guide/topics/security/security.html
55. Apps for Android: http://code.google.com/p/apps-for-android
56. Android DownloadManager: http://developer.android.com/reference/android/app/DownloadManager.html
57. Android File Transfer: http://www.android.com/filetransfer
58. Android Media Formats: http://developer.android.com/guide/appendix/media-formats.html
59. HTTP Live Streaming Draft Protocol: http://tools.ietf.org/html/draft-pantos-http-live-streaming-03
60. NFC Connection Handover: http://www.nfc-forum.org/specs/spec_list/#conn_handover
61. Bluetooth Secure Simple Pairing Using NFC: http://www.nfc-forum.org/resources/AppDocs/NFCForum_AD_BTSSP_1_0.pdf
62. Wifi Multicast API: http://developer.android.com/reference/android/net/wifi/WifiManager.MulticastLock.html
63. Action Assist:

http://developer.android.com/reference/android/content/Intent.html#ACTION_ASSIST

64. USB Charging Specification: http://www.usb.org/developers/devclass_docs/USB_Battery_Charging_1.2.pdf
65. Android Beam: http://developer.android.com/guide/topics/nfc/nfc.html
66. Android USB Audio: http://developer.android.com/reference/android/hardware/usb/UsbConstants.html#USB_CLASS_AUDIO
67. Android NFC Sharing Settings: http://developer.android.com/reference/android/provider/Settings.html#ACTION_NFCSHARING_SETTINGS
68. Wifi Direct (Wifi P2P): http://developer.android.com/reference/android/net/wifi/p2p/WifiP2pManager.html
69. Lock and Home Screen Widget: http://developer.android.com/reference/android/appwidget/AppWidgetProviderInfo.html
70. UserManager reference: http://developer.android.com/reference/android/os/UserManager.html
71. External Storage reference: http://source.android.com/tech/storage
72. External Storage APIs: http://developer.android.com/reference/android/os/Environment.html
73. SMS Short Code: http://en.wikipedia.org/wiki/Short_code
74. Media Remote Control Client: http://developer.android.com/reference/android/media/RemoteControlClient.html
75. Display Manager: http://developer.android.com/reference/android/hardware/display/DisplayManager.html
76. Dreams: http://developer.android.com/reference/android/service/dreams/DreamService.html
77. Android Application Development-Related Settings: http://developer.android.com/reference/android/provider/Settings.html#ACTION_APPLICATION_DEVELOPMENT_SETTINGS
78. Camera: http://developer.android.com/reference/android/hardware/Camera.Parameters.html
79. Motion Event API: http://developer.android.com/reference/android/view/MotionEvent.html
80. Touch Input Configuration: http://source.android.com/tech/input/touch-devices.html

Many of these resources are derived directly or indirectly from the Android 4.2 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 references included above are considered by inclusion to be part of this Compatibility Definition.

3. Software

3.1. Managed API Compatibility

The managed (Dalvik-based) 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 VM environment. Device implementations MUST provide complete implementations, including all documented behaviors, of any documented API exposed by the Android 4.2 SDK [Resources, 4].

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.

3.2. 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 such 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 [Resources, 5]. Note that Section 10 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.Build` class [Resources, 6] 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	Comments
android.os.Build.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 [Resources, 7].
android.os.Build.VERSION.SDK	The version of the currently-executing Android system, in a format accessible to third-party application code. For Android 4.2, this field MUST have the integer value 17.
android.os.Build.VERSION.SDK_INT	The version of the currently-executing Android system, in a format accessible to third-party application code. For Android 4.2, this field MUST have the integer value 17.
android.os.Build.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 re-used 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 ("").
android.os.Build.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.,_-]+$"`.
android.os.Build.BRAND	A value chosen by the device implementer identifying the name of the company, organization, individual, etc. who produced the device, in human-readable format. A possible use of this field is to indicate the OEM and/or carrier who sold the device. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression `"^[a-zA-Z0-9.,_-]+$"`.
android.os.Build.CPU_ABI	The name of the instruction set (CPU type + ABI convention) of native code. See Section 3.3: Native API Compatibility.
android.os.Build.CPU_ABI2	The name of the second instruction set (CPU type + ABI convention) of native code. See Section 3.3: Native API Compatibility.
android.os.Build.DEVICE	A value chosen by the device implementer identifying the specific configuration or revision of the body (sometimes called "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.,_-]+$"`.

android.os.Build.FINGERPRINT	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)`
	For example:
	`acme/mydevice/generic:4.2/JRN53/3359: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.

android.os.Build.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.,_-]+$"`.
android.os.Build.HOST	A string that uniquely identifies the host the build was built on, in human readable format. There are no requirements on the specific format of this field, except that it MUST NOT be null or the empty string ("").
android.os.Build.ID	An identifier chosen by the device implementer to refer to a specific release, in human readable format. This field can be the same as android.os.Build.VERSION.INCREMENTAL, but SHOULD be a value sufficiently meaningful for end users to distinguish between software builds. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression `"^[a-zA-Z0-9.,_-]+$"`.
android.os.Build.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 ("").
android.os.Build.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 ("").
android.os.Build.PRODUCT	A value chosen by the device implementer containing the development name or code name of the product (SKU). 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.,_-]+$"`.
android.os.Build.SERIAL	A hardware serial number, if available. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression `"^([a-zA-Z0-9]{0,20})$"`.
	A comma-separated list of tags chosen by the device implementer that further distinguish the build. For

android.os.Build.TAGS	example, "unsigned,debug". The value of this field MUST be encodable as 7-bit ASCII and match the regular expression `"^[a-zA-Z0-9.,_-]+$"`.
android.os.Build.TIME	A value representing the timestamp of when the build occurred.
android.os.Build.TYPE	A value chosen by the device implementer specifying the runtime configuration of the build. This field SHOULD have one of the values corresponding to the three typical Android runtime configurations: "user", "userdebug", or "eng". The value of this field MUST be encodable as 7-bit ASCII and match the regular expression `"^[a-zA-Z0-9.,_-]+$"`.
android.os.Build.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 ("").

3.2.3. Intent Compatibility

Device implementations MUST honor Android's loose-coupling Intent system, as described in the sections below. By "honored", it is meant that the device implementer MUST provide an Android Activity or Service that specifies a matching Intent filter and binds to and implements correct behavior for each specified Intent pattern.

3.2.3.1. Core Application Intents

The Android upstream project defines a number of core applications, such as contacts, calendar, photo gallery, music player, and so on. Device implementers MAY replace these applications with alternative versions.

However, any such alternative versions MUST honor the same Intent patterns provided by the upstream project. For example, if a device contains an alternative music player, it must still honor the Intent pattern issued by third-party applications to pick a song.

The following applications are considered core Android system applications:

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

The core Android system applications include various Activity, or Service components that are considered "public". That is, the attribute "android:exported" may be absent, or may have the value "true".

For every Activity or Service defined in one of the core Android system apps that is not marked as non-public via an android:exported attribute with the value "false", device implementations MUST include a compontent of the same type implementing the same Intent filter patterns as the core Android system app.

In other words, a device implementation MAY replace core Android system apps; however, if it does, the device implementation MUST support all Intent patterns defined by each core Android system app being replaced.

3.2.3.2. Intent Overrides

As Android is an extensible platform, device implementations MUST allow each Intent pattern referenced in Section 3.2.3.2 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 which allows the user to select between multiple applications which all handle the same Intent pattern.

However, device implementations MAY provide default activities for specific URI patterns (eg. http://play.google.com) if the default activity provides a more specific filter for the data URI. For example, an intent filter specifying the data URI "http://www.android.com" is more specific than the browser filter for "http://". Device implementations MUST provide a user interface for users to modify the default activity for intents.

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.

3.3. Native API Compatibility

3.3.1 Application Binary Interfaces

Managed code running in Dalvik 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, in the file `docs/CPU-ARCH-ABIS.html`. If a device implementation is compatible with one or more defined ABIs, it SHOULD 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 accurately report the native Application Binary Interface (ABI) supported by the device, via the `android.os.Build.CPU_ABI` API
- MUST report only those ABIs documented in the latest version of the Android NDK, in the file `docs/CPU-ARCH-ABIS.txt`
- SHOULD be built using the source code and header files available in the upstream Android open source project

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

- libc (C library)
- libm (math library)
- Minimal support for C++
- JNI interface
- liblog (Android logging)
- libz (Zlib compression)
- libdl (dynamic linker)
- libGLESv1_CM.so (OpenGL ES 1.0)
- libGLESv2.so (OpenGL ES 2.0)
- libEGL.so (native OpenGL surface management)
- libjnigraphics.so
- libOpenSLES.so (OpenSL ES 1.0.1 audio support)
- libOpenMAXAL.so (OpenMAX AL 1.0.1 support)
- libandroid.so (native Android activity support)
- Support for OpenGL, as described below

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 ABI at all.

Native code compatibility is challenging. For this reason, it should be repeated that device implementers are VERY strongly encouraged to use the upstream implementations of the libraries listed above to help ensure compatibility.

3.4. Web Compatibility

3.4.1. WebView Compatibility

The Android Open Source implementation uses the WebKit rendering engine 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 WebKit in the WebView implementation. Specifically:

- Device implementations' `android.webkit.WebView` implementations MUST be based on the 534.30 WebKit build from the upstream Android Open Source tree for Android 4.2. This build includes a specific set of functionality and security fixes for the WebView. Device implementers MAY include customizations to the WebKit implementation; however, any such customizations MUST NOT alter the behavior of the WebView, including rendering behavior.
- The user agent string reported by the WebView MUST be in this format:
  `Mozilla/5.0 (Linux; U; Android $(VERSION); $(LOCALE); $(MODEL) Build/$(BUILD)) AppleWebKit/534.30 (KHTML, like Gecko) Version/4.2 Mobile Safari/534.30`
  - The value of the $(VERSION) string MUST be the same as the value for `android.os.Build.VERSION.RELEASE`
  - The value of the $(LOCALE) string SHOULD follow the ISO conventions for country code and language, and SHOULD refer to the current configured locale of the device
  - 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`
  - Device implementations MAY omit `Mobile` in the user agent string

The WebView component SHOULD include support for as much of HTML5 [Resources, 11] as possible. Minimally, device implementations MUST support each of these APIs associated with HTML5 in the WebView:

- application cache/offline operation [Resources, 12]
- the <video> tag [Resources, 13]
- geolocation [Resources, 14]

Additionally, device implementations MUST support the HTML5/W3C webstorage API [Resources, 15], and SHOULD support the HTML5/W3C IndexedDB API [Resources, 16]. *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.*

HTML5 APIs, like all JavaScript APIs, MUST be disabled by default in a WebView, unless the developer explicitly enables them via the usual Android APIs.

3.4.2. Browser Compatibility

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 [Resources, 11] as possible. Minimally, device implementations MUST support each of these APIs associated with HTML5:

- application cache/offline operation [Resources, 12]
- the <video> tag [Resources, 13]
- geolocation [Resources, 14]

Additionally, device implementations MUST support the HTML5/W3C webstorage API [Resources, 15], and SHOULD support the HTML5/W3C IndexedDB API [Resources, 16]. *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 [Resources, 3]. 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 which 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. Virtual Machine Compatibility

Device implementations MUST support the full Dalvik Executable (DEX) bytecode specification and Dalvik Virtual Machine semantics [Resources, 17].

Device implementations MUST configure Dalvik 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 Size	Screen Density	Application Memory

small / normal / large	ldpi / mdpi	16MB
small / normal / large	tvdpi / hdpi	32MB
small / normal / large	xhdpi	64MB
xlarge	mdpi	32MB
xlarge	tvdpi / hdpi	64MB
xlarge	xhdpi	128MB

3.8. User Interface Compatibility

3.8.1. Widgets

Android defines a component type and corresponding API and lifecycle that allows applications to expose an "AppWidget" to the end user [Resources, 18]. The Android Open Source reference release includes a Launcher application that includes user interface affordances allowing the user to add, view, and remove AppWidgets from the home screen.

Device implementations MAY substitute an alternative to the reference Launcher (i.e. home screen). Alternative Launchers SHOULD include built-in support for AppWidgets, and expose user interface affordances to add, configure, view, and remove AppWidgets directly within the Launcher. Alternative Launchers MAY omit these user interface elements; however, if they are omitted, the device implementation MUST provide a separate application accessible from the Launcher that allows users to add, configure, view, and remove AppWidgets.

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 [Resources, 18] for details.

3.8.2. Notifications

Android includes APIs that allow developers to notify users of notable events [Resources, 19], 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. Note that this behavior is further detailed in Section 7.

Additionally, the implementation MUST correctly render all resources (icons, sound files, etc.) provided for in the APIs [Resources, 20], or in the Status/System Bar icon style guide [Resources, 21]. 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 4.2 includes support for rich notifications, such as interactive Views for ongoing notifications. Device implementations MUST properly display and execute rich notifications, as documented in the Android APIs.

3.8.3. Search

Android includes APIs [Resources, 22] 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.

Device implementations MUST include a single, shared, system-wide search user interface capable of real-time suggestions in response to user input. Device implementations MUST implement the APIs that allow developers to reuse this user interface to provide search within their own applications. Device implementations 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.

3.8.4. Toasts

Applications can use the "Toast" API (defined in [Resources, 23]) 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.

3.8.5. Themes

Android provides "themes" as a mechanism for applications to apply styles across an entire Activity or application. Android 4.2 includes a "Holo" or "holographic" theme 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 [Resources, 24]. Device implementations MUST NOT alter any of the Holo theme attributes exposed to applications [Resources, 25].

Android 4.2 includes a new "Device Default" theme 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 DeviceDefault theme attributes exposed to applications [Resources, 25].

3.8.6. 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 [Resources, 26]. 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 framerate with no adverse affects 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 Open GL 1.0 or 2.0 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. Device implementations determined to not run live wallpapers reliably as described above MUST NOT implement live wallpapers.

3.8.7. Recent Application Display

The upstream Android 4.2 source code includes a user interface for displaying recent applications using a thumbnail image of the application's graphical state at the moment the user last left the application. Device implementations MAY alter or eliminate this user interface; however, a future version of Android is planned to make more extensive use of this functionality. Device implementations are strongly encouraged to use the upstream Android 4.2 user interface (or a similar thumbnail-based interface) for recent applications, or else they may not be compatible with a future version of Android.

3.8.8. Input Management Settings

Android 4.2 includes support for Input Management Engines. The Android 4.2 APIs allow custom app IMEs to specify user-tunable settings. Device implementations MUST include a way for the user to access IME settings at all times when an IME that provides such user settings is displayed.

3.8.9. Lock and Home Screen Widgets

Android 4.2 includes support for application widgets that users can embed in the home screen or the lock screen (See the App Widget Design Guidelines in the Android SDK documentation [Resources, 69] for details). Application widgets allow quick access to application data and services without launching a new activity. Widgets declare support for usage on the home screen or the lock screen by declaring the `android:widgetCategory` manifest tag that tells the system where the widget can be placed. Specifically, device implementations MUST meet the following requirements.

- Device implementations MUST support application widgets on the home screen.
- Device implementations SHOULD support lock screen. If device implementations include support for lock screen then device implementations

MUST support application widgets on the lock screen.

3.8.10. Lock Screen Media Remote Control

Android 4.2 includes support for Remote Control API that lets media applications integrate with playback controls that are displayed in a remote view like the device lock screen[Resources, 74]. Device implementations MUST include support for embedding remote controls in the device lock screen.

3.8.11. Dreams

Android 4.2 includes support for interactive screensavers called Dreams [Resources, 76]. Dreams allows users to interact with applications when a charging device is idle, or docked in a desk dock. Device implementations MUST include support for Dreams and provide a settings option for users to configure Dreams.

3.9 Device Administration

Android 4.2 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 [Resources, 27]. Device implementations MUST provide an implementation of the `DevicePolicyManager` class [Resources, 28], and SHOULD support the full range of device administration policies defined in the Android SDK documentation [Resources, 27].

Note: while some of the requirements outlined above are stated as "SHOULD" for Android 4.2, device implementations that support lock screen MUST support device policies to manage widgets on the lock screen as defined in the Android SDK documentation [Resources, 27].

Note: while some of the requirements outlined above are stated as "SHOULD" for Android 4.2, the Compatibility Definition for a future version is planned to change these to "MUST". That is, these requirements are optional in Android 4.2 but **will be required** by a future version. Existing and new devices that run Android 4.2 are **very strongly encouraged to meet these requirements in Android 4.2**, or they will not be able to attain Android compatibility when upgraded to the future version.

3.10 Accessibility

Android 4.2 provides an accessibility layer that helps users with disabilities to navigate their devices more easily. In addition, Android 4.2 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 [Resources, 29]. Device implementations MUST provide an implementation of the Android accessibility framework consistent with the default Android implementation. Specifically, device implementations MUST meet the following requirements.

- Device implementations MUST support third party accessibility service implementations through the `android.accessibilityservice` APIs [Resources, 30].
- Device implementations MUST generate `AccessibilityEvents` and deliver these events to all registered `AccessibilityService` implementations in a manner consistent with the default Android implementation.
- Device implementations 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.

Additionally, device implementations SHOULD provide an implementation of an accessibility service on the device, and SHOULD provide a mechanism for users to enable the accessibility service during device setup. An open source implementation of an accessibility service is available from the Eyes Free project [Resources, 31].

3.11 Text-to-Speech

Android 4.2 includes APIs that allow applications to make use of text-to-speech (TTS) services, and allows service providers to provide implementations of TTS services [Resources, 32]. Device implementations MUST meet these requirements related to the Android TTS framework:

- 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.
- Device implementations MUST support installation of third-party TTS engines.
- Device implementations MUST provide a user-accessible interface that allows users to select a TTS engine for use at the system level.

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 [Resources, 33].

Devices implementations MUST NOT extend either the .apk [Resources, 34], Android Manifest [Resources, 35], Dalvik bytecode [Resources, 17], or renderscript bytecode formats in such a way that would prevent those files from installing and running correctly on other compatible devices. Device implementers SHOULD use the reference upstream implementation of Dalvik, and the reference implementation's package management system.

5. Multimedia Compatibility

Device implementations MUST include at least one form of audio output, such as speakers, headphone jack, external speaker connection, etc.

5.1. Media Codecs

Device implementations MUST support the core media formats specified in the Android SDK documentation [Resources, 58] except where explicitly permitted in this document. Specifically, device implementations MUST support the media formats, encoders, decoders, file types and container formats defined in the tables below. All of these codecs 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 unencumbered by 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.

Note that these tables do not list specific bitrate requirements for most video codecs because current device hardware does not necessarily support bitrates that map exactly to the required bitrates specified by the relevant standards. Instead, device implementations SHOULD support the highest bitrate practical on the hardware, up to the limits defined by the specifications.

Type	Format / Codec	Encoder	Decoder	Details	File Type(s) / Container Formats
Audio	MPEG-4 AAC Profile (AAC LC)	REQUIRED Required for device implementations that include microphone hardware and define `android.hardware.microphone`.	REQUIRED	Support for mono/stereo/5.0/5.1* 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 for device implementations that include microphone hardware and define android.hardware.microphone	REQUIRED	Support for mono/stereo/5.0/5.1* content with standard sampling rates from 16 to 48 kHz.	
	MPEG-4 HE AAC v2 Profile (enhanced AAC+)		REQUIRED	Support for mono/stereo/5.0/5.1* content with standard sampling rates from 16 to 48 kHz.	
	MPEG-4 Audio Object Type ER AAC ELD (Enhanced Low Delay AAC)	REQUIRED for device implementations that include microphone hardware and define android.hardware.microphone	REQUIRED	Support for mono/stereo content with standard sampling rates from 16 to 48 kHz.	
	AMR-NB	REQUIRED Required for device implementations that include microphone hardware and define `android.hardware.microphone`.	REQUIRED	4.75 to 12.2 kbps sampled @ 8kHz	3GPP (.3gp)
	AMR-WB	REQUIRED Required for device implementations that include microphone hardware and define `android.hardware.microphone`.	REQUIRED	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16kHz	3GPP (.3gp)
	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 bit-rate (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 (.rttl, .rtx) • OTA (.ota) • iMelody (.imy)

	Vorbis		REQUIRED		• Ogg (.ogg) • Matroska (.mkv)
	PCM/WAVE	REQUIRED	REQUIRED	8-bit and 16-bit linear PCM** (rates up to limit of hardware).Devices MUST support sampling rates for raw PCM recording at 8000,16000 and 44100 Hz frequencies	WAVE (.wav)
Image	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)
Video	H.263	REQUIRED Required for device implementations that include camera hardware and define `android.hardware.camera` or `android.hardware.camera.front`.	REQUIRED		• 3GPP (.3gp) • MPEG-4 (.mp4)
	H.264 AVC	REQUIRED Required for device implementations that include camera hardware and define `android.hardware.camera` or `android.hardware.camera.front`.	REQUIRED	Baseline Profile (BP)	• 3GPP (.3gp) • MPEG-4 (.mp4) • MPEG-TS (.ts, AAC audio only, not seekable, Android 3.0+)
	MPEG-4 SP		REQUIRED		3GPP (.3gp)
	VP8		REQUIRED (Android 2.3.3+)		WebM (.webm) and Matroska (.mkv, Android 4.0+)

*Note: Only downmix of 5.0/5.1 content is required; recording or rendering more than 2 channels is optional. **Note: 16-bit linear PCM capture is mandatory. 8-bit linear PCM capture is not mandatory.

5.2 Video Encoding

Android device implementations that include a rear-facing camera and declare `android.hardware.camera` SHOULD support the following video encoding profiles.

	SD (Low quality)	SD (High quality)	HD (When supported by hardware)
Video codec	H.264 Baseline Profile	H.264 Baseline Profile	H.264 Baseline Profile
Video resolution	176 x 144 px	480 x 360 px	1280 x 720 px
Video frame rate	12 fps	30 fps	30 fps
Video bitrate	56 Kbps	500 Kbps or higher	2 Mbps or higher
Audio codec	AAC-LC	AAC-LC	AAC-LC

Audio channels	1 (mono)	2 (stereo)	2 (stereo)
Audio bitrate	24 Kbps	128 Kbps	192 Kbps

5.3 Video Decoding

Android device implementations SHOULD support the following VP8 video decoding profiles.

	SD (Low quality)	SD (High quality)	HD 720p (When supported by hardware)	HD 1080p (When supported by hardware)
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	8 Mbps	20 Mbps

5.4. Audio Recording

When an application has used the `android.media.AudioRecord` API to start recording an audio stream, device implementations that include microphone hardware and declare `android.hardware.microphone` MUST sample and record audio with each of these behaviors:

- 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 1Khz at 90 dB SPL input level.

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:

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

Note: while some of the requirements outlined above are stated as "SHOULD" for Android 4.2, the Compatibility Definition for a future version is planned to change these to "MUST". That is, these requirements are optional in Android 4.2 but **will be required** by a future version. Existing and new devices that run Android 4.2 are **very strongly encouraged to meet these requirements in Android 4.2**, or they will not be able to attain Android compatibility when upgraded to the future version.

5.5. 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 effects such sound effects or VOIP communication.

For the purposes of this section:

- "output latency" is defined as the interval between when an application writes a frame of PCM-coded data and when the corresponding sound can be heard by an external listener or observed by a transducer
- "cold output latency" is defined as 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" is defined as the output latency for subsequent frames, after the device is already playing audio
- "input latency" is the interval between when an external sound is presented to the device and when an application reads the corresponding frame of PCM-coded data
- "cold input latency" is defined as 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" is defined as the input latency for subsequent frames, while the device is already capturing audio
- "OpenSL ES PCM buffer queue API" is the set of PCM-related OpenSL ES APIs within Android NDK; see *NDK_root*`/docs/opensles/index.html`

Per Section 5, all compatible device implementations MUST include at least one form of audio output. Device implementations SHOULD meet or exceed these output latency requirements:

- cold output latency of 100 milliseconds or less
- continuous output latency of 45 milliseconds or less

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 MAY report support for low-latency audio, by reporting the feature "android.hardware.audio.low-latency" via the `android.content.pm.PackageManager` class. [Resources, 37] Conversely, if the device implementation does not meet these requirements it MUST NOT report support for low-latency audio.

Per Section 7.2.5, microphone hardware may be omitted by device implementations.

Device implementations that include microphone hardware and declare `android.hardware.microphone` SHOULD meet these input audio latency requirements:

- cold input latency of 100 milliseconds or less
- continuous input latency of 50 milliseconds or less

5.6. Network Protocols

Devices MUST support the media network protocols for audio and video playback as specified in the Android SDK documentation [Resources, 58]. Specifically, devices MUST support the following media network protocols:

- RTSP (RTP, SDP)
- HTTP(S) progressive streaming
- HTTP(S) Live Streaming draft protocol, Version 3 [Resources, 59]

6. Developer Tools and Options Compatibility

6.1 Developer Tools

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

- **Android Debug Bridge (known as adb)** [Resources, 33]
  Device implementations MUST support all `adb` functions as documented in the Android SDK. 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.

  Android 4.2.2 includes support for secure adb. Secure adb enables adb on known authenticated hosts. Existing and new devices that run Android 4.2.2 are **very strongly encouraged to meet this requirement in Android 4.2** , or they will not be able to attain Android compatibility when upgraded to the future version.

- **Dalvik Debug Monitor Service (known as ddms)** [Resources, 33]
  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** [Resources, 36]
  Device implementations MUST include the Monkey framework, and make it available for applications to use.
- **SysTrace** [Resources, 33]
  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, and Windows 8, in both 32-bit and 64-bit versions.

6.2 Developer Options

Android 4.2 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 [Resources, 77]. 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.

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's APIs MUST still be present
- 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 accurately report accurate hardware configuration information via the `getSystemAvailableFeatures()` and `hasSystemFeature(String)` methods on the `android.content.pm.PackageManager` class. [Resources, 37]

7.1. Display and Graphics

Android 4.2 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 [Resources, 38]. 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" is the distance in inches between two opposing corners of the illuminated portion of the display.
- "dpi" (meaning "dots per inch") is 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" is the ratio of the longer dimension of the screen to the shorter dimension. For example, a display of 480x854 pixels would be 854 / 480 = 1.779, or roughly "16:9".
- A "density-independent pixel" or ("dp") is the virtual pixel unit normalized to a 160 dpi screen, calculated as: `pixels = dps * (density / 160)`.

7.1.1. Screen Configuration

Screen Size

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 [Resources, 38] 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')
- 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, devices MUST have screen sizes of 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.

Screen Aspect Ratio

The aspect ratio MUST be between 1.3333 (4:3) and 1.85 (16:9).

Screen Density

The Android UI framework defines a set of standard logical densities to help application developers target application resources. Device implementations MUST report one of the following logical Android framework densities through the `android.util.DisplayMetrics` APIs, and MUST execute applications at this standard density.

- 120 dpi, known as 'ldpi'
- 160 dpi, known as 'mdpi'
- 213 dpi, known as 'tvdpi'
- 240 dpi, known as 'hdpi'
- 320 dpi, known as 'xhdpi'
- 480 dpi, known as 'xxhdpi'

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.

7.1.2. Display Metrics

Device implementations MUST report correct values for all display metrics defined in `android.util.DisplayMetrics` [Resources, 39].

7.1.3. Screen Orientation

Devices 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.

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, MUST only report `android.hardware.screen.landscape`.

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 MUST also support Android Renderscript, as detailed in the Android SDK documentation [Resources, 8].

Device implementations MUST also correctly identify themselves as supporting OpenGL ES 1.0 and 2.0. That is:

- The managed APIs (such as via the `GLES10.getString()` method) MUST report support for OpenGL ES 1.0 and 2.0

- The native C/C++ OpenGL APIs (that is, those available to apps via libGLES_v1CM.so, libGLES_v2.so, or libEGL.so) MUST report support for OpenGL ES 1.0 and 2.0.

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 4.2 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 4.2 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 4.2 includes a mechanism for applications to declare that they wanted 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 [Resources, 9].

In Android 4.2, 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 [Resources, 9].

Android 4.2 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.

7.1.5. Legacy Application Compatibility Mode

Android 4.2 specifies a "compatibility mode" in which the framework operates in an '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. 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 Types

Device implementation screens are classified as one of two types:

- Fixed-pixel display implementations: the screen is a single panel that supports only a single pixel width and height. Typically the screen is physically integrated with the device. Examples include mobile phones, tablets, and so on.
- Variable-pixel display implementations: the device implementation either has no embedded screen and includes a video output port such as VGA, HDMI or a wireless port for display, or has an embedded screen that can change pixel dimensions. Examples include televisions, set-top boxes, and so on.

Fixed-Pixel Device Implementations

Fixed-pixel device implementations MAY use screens of any pixel dimensions, provided that they meet the requirements defined this Compatibility Definition.

Fixed-pixel implementations MAY include a video output port for use with an external display. However, if that display is ever used for running apps, the device MUST meet the following requirements:

- The device MUST report the same screen configuration and display metrics, as detailed in Sections 7.1.1 and 7.1.2, as the fixed-pixel display.
- The device MUST report the same logical density as the fixed-pixel display.
- The device MUST report screen dimensions that are the same as, or very close to, the fixed-pixel display.

For example, a tablet that is 7" diagonal size with a 1024x600 pixel resolution is considered a fixed-pixel large mdpi display implementation. If it contains a video output port that displays at 720p or 1080p, the device implementation MUST scale the output so that applications are only executed in a large mdpi window, regardless of

whether the fixed-pixel display or video output port is in use.

Variable-Pixel Device Implementations

Variable-pixel device implementations MUST support one or both of 1280x720, or 1920x1080 (that is, 720p or 1080p). Device implementations with variable-pixel displays MUST NOT support any other screen configuration or mode. Device implementations with variable-pixel screens MAY change screen configuration or mode at runtime or boot-time. For example, a user of a set-top box may replace a 720p display with a 1080p display, and the device implementation may adjust accordingly.

Additionally, variable-pixel device implementations MUST report the following configuration buckets for these pixel dimensions:

- 1280x720 (also known as 720p): 'large' screen size, 'tvdpi' (213 dpi) density
- 1920x1080 (also known as 1080p): 'large' screen size, 'xhdpi' (320 dpi) density

For clarity, device implementations with variable pixel dimensions are restricted to 720p or 1080p in Android 4.2, and MUST be configured to report screen size and density buckets as noted above.

7.1.7. 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. Specifically:

- 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 1.1. That is, the pixel aspect ratio MUST be near square (1.0) with a 10% tolerance.

7.1.8. External Displays

Android 4.2 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 [Resources, 75]. Device implementations that support secure video output and are capable of supporting secure surfaces MUST declare support for `Display.SECURE_FLAG`. Specifically, device implementations that declare support for `Display.SECURE_FLAG`, MUST support **HDCP 2.x or higher** for Miracast wireless displays or **HDCP 1.2 or higher** for wired displays. The upstream Android open source implementation includes support for wireless (Miracast) and wired (HDMI) displays that satisfies this requirement.

7.2. Input Devices

7.2.1. Keyboard

Device implementations:

- MUST include support for the Input Management Framework (which allows third party developers to create Input Management Engines - 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)
- 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` [Resources, 40] (that is, QWERTY, or 12-key)

7.2.2. Non-touch Navigation

Device implementations:

- MAY omit a non-touch navigation option (that is, may omit a trackball, d-pad, or wheel)
- MUST report the correct value for `android.content.res.Configuration.navigation` [Resources, 40]
- 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 Home, Menu and Back functions are essential to the Android navigation paradigm. Device implementations MUST make these functions available to the user at all times when running applications. These functions MAY be implemented via dedicated physical buttons (such as mechanical or capacitive touch buttons), or MAY be implemented using dedicated software keys, gestures, touch panel, etc. Android 4.2 supports both implementations.

Android 4.2 includes support for assist action [Resources, 63]. Device implementations MUST make the assist action available to the user at all times when running applications. This function MAY be implemented via hardware or software keys.

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`.
- Device implementation MUST present a Menu key to applications when targetSdkVersion <= 10 and SHOULD NOT present a Menu key when the targetSdkVersion > 10.

7.2.4. 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` [Resources, 40] corresponding to the type of the specific touchscreen on the device

Android 4.0 includes support for a variety of touch screens, touch pads, and fake touch input devices. Touch screen based device implementations are associated with a display [Resources, 80] 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 4.0 includes the feature constant `android.hardware.faketouch`, which corresponds to a high-fidelity non-touch (that is, 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[Resources, 79]
- MUST report touch event with the action code [Resources, 79] that specifies the state change that occurs on the pointer going down or up on the screen [Resources, 79]
- 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 double tap on an object on the screen [Resources, 79]
- 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. 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 as no-ops, per Section 7. Conversely, device implementations that do possess a microphone:

- MUST report the android.hardware.microphone feature constant
- SHOULD meet the audio quality requirements in Section 5.4
- SHOULD meet the audio latency requirements in Section 5.5

7.3. Sensors

Android 4.2 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. For example, device implementations:

- MUST accurately report the presence or absence of sensors per the android.content.pm.PackageManager class. [Resources, 37]
- 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 (i.e. metric) values for each sensor type as defined in the Android SDK documentation [Resources, 41]

The list above is not comprehensive; the documented behavior of the Android SDK is to be considered authoritative.

Some sensor types are synthetic, 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.

The Android 4.2 includes a notion of a "streaming" sensor, which is one that returns data continuously, rather than only when the data changes. Device implementations MUST continuously provide periodic data samples for any API indicated by the Android 4.2 SDK documentation to be a streaming sensor. Note that the device implementations MUST ensure that the sensor stream must not prevent the device

CPU from entering a suspend state or waking up from a suspend state.

7.3.1. Accelerometer

Device implementations SHOULD include a 3-axis accelerometer. If a device implementation does include a 3-axis accelerometer, it:

- SHOULD be able to deliver events at 120 Hz or greater. Note that while the accelerometer frequency above is stated as "SHOULD" for Android 4.2, the Compatibility Definition for a future version is planned to change these to "MUST". That is, these standards are optional in Android 4.2 but **will be required** in future versions. Existing and new devices that run Android 4.2 are **very strongly encouraged to meet these requirements in Android 4.2** so they will be able to upgrade to the future platform releases
- MUST comply with the Android sensor coordinate system as detailed in the Android APIs (see [Resources, 41])
- MUST be capable of measuring from freefall up to twice gravity (2g) or more on any three-dimensional vector
- MUST have 8-bits of accuracy or more
- MUST have a standard deviation no greater than 0.05 m/s^2

7.3.2. Magnetometer

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

- MUST be able to deliver events at 10 Hz or greater
- MUST comply with the Android sensor coordinate system as detailed in the Android APIs (see [Resources, 41]).
- MUST be capable of sampling a range of field strengths adequate to cover the geomagnetic field
- MUST have 8-bits of accuracy or more
- MUST have a standard deviation no greater than 0.5 µT

7.3.3. GPS

Device implementations SHOULD include a GPS receiver. If a device implementation does include a GPS receiver, it SHOULD include some form of "assisted GPS" technique to minimize GPS lock-on time.

7.3.4. Gyroscope

Device implementations SHOULD include a gyroscope (i.e. 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 be temperature compensated
- MUST be capable of measuring orientation changes up to 5.5*Pi radians/second (that is, approximately 1,000 degrees per second)
- SHOULD be able to deliver events at 200 Hz or greater. Note that while the gyroscope frequency above is stated as "SHOULD" for Android 4.2, the Compatibility Definition for a future version is planned to change these to "MUST". That is, these standards are optional in Android 4.2 but **will be required** in future versions. Existing and new devices that run Android 4.2 are **very strongly encouraged to meet these requirements in Android 4.2** so they will be able to upgrade to the future platform releases
- MUST have 12-bits of accuracy or more
- 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 have timestamps as close to when the hardware event happened as possible. The constant latency must be removed.

7.3.5. Barometer

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

- MUST be able to deliver events at 5 Hz or greater
- MUST have adequate precision to enable estimating altitude
- MUST be temperature compensated

7.3.7. Thermometer

Device implementations MAY but SHOULD NOT include a thermometer (i.e. temperature sensor.) If a device implementation does include a thermometer, it MUST measure the temperature of the device CPU. It MUST NOT measure any other temperature. (Note that this sensor type is deprecated in the Android 4.2 APIs.)

7.3.7. Photometer

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

7.3.8. Proximity Sensor

Device implementations MAY 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. If a device implementation has a proximity sensor, it MUST be have 1-bit of accuracy or more.

7.4. Data Connectivity

7.4.1. Telephony

"Telephony" as used by the Android 4.2 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 4.2 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 sub-features, regardless of whether they use a cellular network for data connectivity.

Android 4.2 MAY be used on devices that do not include telephony hardware. That is, Android 4.2 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.

7.4.2. IEEE 802.11 (WiFi)

Android 4.2 device implementations SHOULD include support for one or more forms of 802.11 (b/g/a/n, etc.) If a device implementation does include support for 802.11, it MUST implement the corresponding Android API.

Device implementations MUST implement the multicast API as described in the SDK documentation [Resources, 62]. Device implementations that do include Wifi support MUST support multicast DNS (mDNS). Device implementations MUST not filter mDNS packets (224.0.0.251) at any time of operation including when the screen is not in an active state.

7.4.2.1. WiFi Direct

Device implementations SHOULD include support for Wifi direct (Wifi peer-to-peer). If a device implementation does include support for Wifi direct, it MUST implement the corresponding Android API as described in the SDK documentation [Resources, 68]. If a device implementation includes support for Wifi direct, then it:

- MUST support regular Wifi operation
- SHOULD support concurrent wifi and wifi Direct operation

7.4.3. Bluetooth

Device implementations SHOULD include a Bluetooth transceiver. Device implementations that do include a Bluetooth transceiver MUST enable the RFCOMM-based Bluetooth API as described in the SDK documentation [Resources, 42]. Device implementations SHOULD implement relevant Bluetooth profiles, such as A2DP, AVRCP, OBEX, etc. as appropriate for the device.

The Compatibility Test Suite includes cases that cover basic operation of the Android RFCOMM Bluetooth API. However, since Bluetooth is a communications protocol between devices, it cannot be fully tested by unit tests running on a single device. Consequently, device implementations MUST also pass the human-driven Bluetooth test procedure described in Appendix A.

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, then it:

- MUST report the android.hardware.nfc feature from the `android.content.pm.PackageManager.hasSystemFeature()` method. [Resources, 37]
- 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 6319-4)
    - IsoDep (ISO 14443-4)
    - NFC Forum Tag Types 1, 2, 3, 4 (defined by the NFC Forum)
- SHOULD be capable of reading and writing NDEF messages via the following NFC standards. Note that while the NFC standards below are stated as "SHOULD" for Android 4.2, the Compatibility Definition for a future version is planned to change these to "MUST". That is, these standards are optional in Android 4.2 but **will be required** in future versions. Existing and new devices that run Android 4.2 are **very strongly encouraged to meet these requirements in Android 4.2** so they will be able to upgrade to the future platform releases.
  - NfcV (ISO 15693)
- MUST be capable of transmitting and receiving data via the following peer-to-peer standards and protocols:
  - ISO 18092
  - LLCP 1.0 (defined by the NFC Forum)
  - SDP 1.0 (defined by the NFC Forum)
  - NDEF Push Protocol [Resources, 43]
  - SNEP 1.0 (defined by the NFC Forum)
- MUST include support for Android Beam [Resources, 65]:
  - 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.
  - Device implementations MUST honor the android.settings.NFCSHARING_SETTINGS intent to show NFC sharing settings [Resources, 67].
  - 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
  - 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" [Resources, 60] and "Bluetooth Secure Simple Pairing Using NFC version 1.0" [Resources, 61] specs from the NFC Forum. Such an implementation SHOULD use SNEP GET requests for exchanging the handover request / select records over NFC, and it MUST use the Bluetooth Object Push Profile for the actual Bluetooth data transfer.
- 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.)

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

- MIFARE Classic (NXP MF1S503x [Resources, 44], MF1S703x [Resources, 44])
- MIFARE Ultralight (NXP MF0ICU1 [Resources, 46], MF0ICU2 [Resources, 46])
- NDEF on MIFARE Classic (NXP AN130511 [Resources, 48], AN130411 [Resources, 49])

Note that Android 4.2 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. [Resources, 37] Note that this is not a standard Android feature, and as such does not appear as a constant on the `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 [Resources, 37], and MUST implement the Android 4.2 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, 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 (WiFi).

Devices MAY implement more than one form of data connectivity.

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.

7.5.1. Rear-Facing Camera

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

- 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 a front-facing camera, it:

- MUST have a resolution of at least VGA (that is, 640x480 pixels)
- MUST NOT use a front-facing camera as the default for the Camera API. That is, the camera API in Android 4.2 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()` [Resources, 50] 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. Camera API Behavior

Device implementations MUST implement the following behaviors for the camera-related APIs, for both front- and rear-facing cameras:

1. 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.
2. 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.
3. 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.)

Device implementations MUST implement the full Camera API included in the Android 4.2 SDK documentation [Resources, 51]), 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` [Resources, 78]).

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.

7.5.4. 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 dimention. 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

Device implementations MUST have at least 340MB of memory available to the kernel and userspace. The 340MB MUST be in addition to any memory dedicated to hardware components such as radio, video, and so on that is not under the kernel's control.

Device implementations MUST have at least 350MB of non-volatile storage available for application private data. That is, the `/data` partition MUST be at least 350MB.

The Android APIs include a Download Manager that applications may use to download data files [Resources, 56]. 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. The shared storage provided MUST be at least 1GB in size.

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

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 MAY have hardware for user-accessible removable storage, such as a Secure Digital card. Alternatively, device implementations MAY allocate internal (non-removable) storage as shared storage for apps.

Regardless of the form of shared storage used, device implementations MUST provide some mechanism to access the contents of shared storage from a host computer, such as USB mass storage (UMS) or Media Transfer Protocol (MTP). Device implementations MAY use USB mass storage, but SHOULD use Media Transfer Protocol. If the device implementation supports Media Transfer Protocol:

- The device implementation SHOULD be compatible with the reference Android MTP host, Android File Transfer [Resources, 57].
- The device implementation SHOULD report a USB device class of `0x00`.
- The device implementation SHOULD report a USB interface name of 'MTP'.

If the device implementation lacks USB ports, it MUST provide a host computer with access to the contents of shared storage by some other means, such as a network file system.

It is illustrative to consider two common examples. If a device implementation includes an SD card slot to satisfy the shared storage requirement, a FAT-formatted SD card 1GB in size or larger MUST be included with the device as sold to users, and MUST be mounted by default. Alternatively, if a device implementation uses internal fixed storage to satisfy this requirement, that storage MUST be 1GB in size or larger and mounted on `/sdcard` (or `/sdcard` MUST be a symbolic link to the physical location if it is mounted elsewhere.)

Device implementations that include multiple shared storage paths (such as both an SD card slot and shared internal storage) SHOULD modify the core applications such as the media scanner and ContentProvider to transparently support files placed in both

locations.

7.7. USB

Device implementations SHOULD include a USB client port, and SHOULD include a USB host port.

If a device implementation includes a USB client port:

- the port MUST be connectable to a USB host with a standard USB-A port
- the port SHOULD use the micro USB form factor on the device side. Existing and new devices that run Android 4.2 are **very strongly encouraged to meet these requirements in Android 4.2** so they will be able to upgrade to the future platform releases
- the port SHOULD be centered in the middle of an edge. Device implementations SHOULD either locate the port 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 devices that run Android 4.2 are **very strongly encouraged to meet these requirements in Android 4.2** so they will be able to upgrade to future platform releases.
- if the device has other ports (such as a non-USB charging port) it SHOULD be on the same edge as the micro-USB port
- it MUST allow a host connected to the device to access the contents of the shared storage volume using either USB mass storage or Media Transfer Protocol
- it MUST implement the Android Open Accessory API and specification as documented in the Android SDK documentation, and MUST declare support for the hardware feature `android.hardware.usb.accessory` [Resources, 52]
- it MUST implement the USB audio class as documented in the Android SDK documentation [Resources, 66]
- it SHOULD implement support for USB battery charging specification [Resources, 64] Existing and new devices that run Android 4.2 are **very strongly encouraged to meet these requirements in Android 4.2** so they will be able to upgrade to the future platform releases

If a device implementation includes a USB host port:

- it MAY use a non-standard port form factor, but if so MUST ship with a cable or cables adapting the port to standard USB-A
- it 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` [Resources, 53]

Device implementations MUST implement the Android Debug Bridge. If a device implementation omits a USB client port, it MUST implement the Android Debug Bridge via local-area network (such as Ethernet or 802.11)

8. Performance Compatibility

Device implementations MUST meet the key performance metrics of an Android 4.2 compatible device defined in the table below:

Metric	Performance Threshold	Comments
Application Launch Time	The following applications should launch within the specified time. • Browser: less than 1300ms • Contacts: less than 700ms • Settings: less than 700ms	The launch time is measured as the total time to complete loading the default activity for the application, including the time it takes to start the Linux process, load the Android package into the Dalvik VM, and call onCreate.
Simultaneous Applications	When multiple applications have been launched, re-launching an already-running application after it has been launched must take less than the original launch time.	

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 [Resources, 54] 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 sub-sections.

9.1. Permissions

Device implementations MUST support the Android permissions model as defined in the Android developer documentation [Resources, 54]. 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.

9.2. UID and Process Isolation

Device implementations MUST support the Android application sandbox model, in which each application runs as a unique Unix-style 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 [Resources, 54].

9.3. Filesystem Permissions

Device implementations MUST support the Android file access permissions model as defined in as defined in the Security and Permissions reference [Resources, 54].

9.4. Alternate Execution Environments

Device implementations MAY include runtime environments that execute applications using some other software or technology than the Dalvik virtual machine 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 (that is, Linux user IDs, etc.)
- Alternate runtimes MAY provide a single Android sandbox shared by all applications using the alternate runtime
- Alternate runtimes and 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
- Alternate runtimes MUST NOT launch with, grant, or be granted access to the sandboxes corresponding to other Android applications

Alternate runtimes 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. That is, 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

Android 4.2 includes support for multiple users and provides support for full user isolation [Resources, 70].

Device implementations MUST meet these requirements related to multi-user support[Resources, 71]:

- As the behavior of the telephony APIs on devices with multiple users is currently undefined, device implementations that declare android.hardware.telephony MUST NOT enable multi-user support.
- 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 [Resources, 54]

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 multi-user 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 [Resources, 72] for primary external storage. The upstream Android open-source project includes an implementation that uses internal device storage for application external storage APIs; device implementations SHOULD use this configuration and software implementation. Device implementations that include multiple external storage paths MUST NOT allow Android applications to write to the secondary external storage

9.6. Premium SMS Warning

Android 4.2 includes support for warning users for 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.

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 very strongly encouraged to make the minimum number of changes as possible to the reference and preferred implementation of Android 4.2 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) [Resources, 2] 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 4.2. 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 which 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 Verfier only by the set of included locales, branding, etc. MAY omit the CTS Verifier test.

10.3. Reference Applications

Device implementers MUST test implementation compatibility using the following open source applications:

- The "Apps for Android" applications [Resources, 55]
- Replica Island (available in Android Market)

Each app above MUST launch and behave correctly on the implementation, for the implementation to be considered compatible.

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

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.

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.

12. Contact Us

You can contact the document authors at compatibility@android.com for clarifications and to bring up any issues that you think the document does not cover.

Appendix A - Bluetooth Test Procedure

The Compatibility Test Suite includes cases that cover basic operation of the Android RFCOMM Bluetooth API. However, since Bluetooth is a communications protocol between devices, it cannot be fully tested by unit tests running on a single device. Consequently, device implementations MUST also pass the human-operated Bluetooth test procedure described below.

The test procedure is based on the BluetoothChat sample app included in the Android open source project tree. The procedure requires two devices:

- a candidate device implementation running the software build to be tested
- a separate device implementation already known to be compatible, and of a model from the device implementation being tested - that is, a "known good" device implementation

The test procedure below refers to these devices as the "candidate" and "known good" devices, respectively.

Setup and Installation

1. Build BluetoothChat.apk via 'make samples' from an Android source code tree
2. Install BluetoothChat.apk on the known-good device
3. Install BluetoothChat.apk on the candidate device

Test Bluetooth Control by Apps

1. Launch BluetoothChat on the candidate device, while Bluetooth is disabled
2. Verify that the candidate device either turns on Bluetooth, or prompts the user with a dialog to turn on Bluetooth

Test Pairing and Communication

1. Launch the Bluetooth Chat app on both devices
2. Make the known-good device discoverable from within BluetoothChat (using the Menu)
3. On the candidate device, scan for Bluetooth devices from within BluetoothChat (using the Menu) and pair with the known-good device
4. Send 10 or more messages from each device, and verify that the other device receives them correctly
5. Close the BluetoothChat app on both devices by pressing **Home**
6. Unpair each device from the other, using the device Settings app

Test Pairing and Communication in the Reverse Direction

1. Launch the Bluetooth Chat app on both devices.
2. Make the candidate device discoverable from within BluetoothChat (using the Menu).
3. On the known-good device, scan for Bluetooth devices from within BluetoothChat (using the Menu) and pair with the candidate device.
4. Send 10 or messages from each device, and verify that the other device receives them correctly.
5. Close the Bluetooth Chat app on both devices by pressing Back repeatedly to get to the Launcher.

Test Re-Launches

1. Re-launch the Bluetooth Chat app on both devices.
2. Send 10 or messages from each device, and verify that the other device receives them correctly.

Note: the above tests have some cases which end a test section by using Home, and some using Back. These tests are not redundant and are not optional: the objective is to verify that the Bluetooth API and stack works correctly both when Activities are explicitly terminated (via the user pressing Back, which calls finish()), and implicitly sent to background (via the user pressing Home.) Each test sequence MUST be performed as described.