



Android 2.2 Compatibility Definition

Copyright © 2010, Google Inc. All rights reserved.

compatibility@android.com

Table of Contents

[1. Introduction](#)

[2. Resources](#)

[3. Software](#)

[3.1. Managed API Compatibility](#)

[3.2. Soft API Compatibility](#)

[3.2.1. Permissions](#)

[3.2.2. Build Parameters](#)

[3.2.3. Intent Compatibility](#)

[3.2.3.1. Core Application Intents](#)

[3.2.3.2. Intent Overrides](#)

[3.2.3.3. Intent Namespaces](#)

[3.2.3.4. Broadcast Intents](#)

[3.3. Native API Compatibility](#)

[3.4. Web Compatibility](#)

[3.4.1. WebView Compatibility](#)

[3.4.2. Browser Compatibility](#)

[3.5. API Behavioral Compatibility](#)

[3.6. API Namespaces](#)

[3.7. Virtual Machine Compatibility](#)

[3.8. User Interface Compatibility](#)

[3.8.1. Widgets](#)

[3.8.2. Notifications](#)

[3.8.3. Search](#)

[3.8.4. Toasts](#)

[3.8.5. Live Wallpapers](#)

[4. Reference Software Compatibility](#)

[5. Application Packaging Compatibility](#)

[6. Multimedia Compatibility](#)

[6.1. Media Codecs](#)

[6.2. Audio Recording](#)

[6.3. Audio Latency](#)

[7. Developer Tool Compatibility](#)

[8. Hardware Compatibility](#)

[8.1. Display](#)

[8.1.2. Non-Standard Display Configurations](#)

[8.1.3. Display Metrics](#)

[8.1.4. Declared Screen Support](#)

[8.2. Keyboard](#)

[8.3. Non-touch Navigation](#)

[8.4. Screen Orientation](#)

[8.5. Touchscreen input](#)

[8.6. USB](#)

[8.7. Navigation keys](#)

[8.8. Wireless Data Networking](#)

[8.9. Camera](#)

[8.10. Accelerometer](#)

[8.11. Compass](#)

[8.12. GPS](#)

[8.13. Telephony](#)

[8.14. Memory and Storage](#)

[8.15. Application Shared Storage](#)

[8.16. Bluetooth](#)

[9. Performance Compatibility](#)

[10. Security Model Compatibility](#)

[10.1. Permissions](#)

[10.2. UID and Process Isolation](#)

[10.3. Filesystem Permissions](#)

[10.4. Alternate Execution Environments](#)

[11. Compatibility Test Suite](#)

[12. Updatable Software](#)

[13. Contact Us](#)

[Appendix A - Bluetooth Test Procedure](#)

1. Introduction

This document enumerates the requirements that must be met in order for mobile phones to be compatible with Android 2.2.

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 2.2. A "device implementation" or "implementation" is the hardware/software solution so developed.

To be considered compatible with Android 2.2, device implementations:

- MUST meet the requirements presented in this Compatibility Definition, including any documents incorporated via reference.
- MUST pass the most recent version of the Android Compatibility Test Suite (CTS) available at the time of the device implementation's software is completed. (The CTS is available as part of the Android Open Source Project [Resources, 2].) The CTS tests many, but not all, of the components outlined in this document.

Where this definition or the CTS 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 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 CTS 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 2.2 allowed version strings: <http://source.android.com/compatibility/2.2/versions.html>
8. android.webkit.WebView class: <http://developer.android.com/reference/android/webkit/WebView.html>
9. HTML5: <http://www.whatwg.org/specs/web-apps/current-work/multipage/>
10. Dalvik Virtual Machine specification: available in the Android source code, at dalvik/docs
11. AppWidgets: http://developer.android.com/guide/practices/ui_guidelines/widget_design.html
12. Notifications: <http://developer.android.com/guide/topics/ui/notifiers/notifications.html>
13. Application Resources: <http://code.google.com/android/reference/available-resources.html>
14. Status Bar icon style guide: http://developer.android.com/guide/practices/ui_guideline/icon_design.html#statusbarstructure
15. Search Manager: <http://developer.android.com/reference/android/app/SearchManager.html>
16. Toasts: <http://developer.android.com/reference/android/widget/Toast.html>
17. Live Wallpapers: <http://developer.android.com/resources/articles/live-wallpapers.html>
18. Apps for Android: <http://code.google.com/p/apps-for-android>
19. Reference tool documentation (for adb, aapt, ddms): <http://developer.android.com/guide/developing/tools/index.html>
20. Android apk file description: <http://developer.android.com/guide/topics/fundamentals.html>
21. Manifest files: <http://developer.android.com/guide/topics/manifest/manifest-intro.html>
22. Monkey testing tool: <http://developer.android.com/guide/developing/tools/monkey.html>
23. Android Hardware Features List: <http://developer.android.com/reference/android/content/pm/PackageManager.html>
24. Supporting Multiple Screens: http://developer.android.com/guide/practices/screens_support.html
25. android.content.res.Configuration: <http://developer.android.com/reference/android/content/res/Configuration.html>

26. android.util.DisplayMetrics: <http://developer.android.com/reference/android/util/DisplayMetrics.html>
27. android.hardware.Camera: <http://developer.android.com/reference/android/hardware/Camera.html>
28. Sensor coordinate space: <http://developer.android.com/reference/android/hardware/SensorEvent.html>
29. Android Security and Permissions reference: <http://developer.android.com/guide/topics/security/security.html>
30. Bluetooth API: <http://developer.android.com/reference/android/bluetooth/package-summary.html>

Many of these resources are derived directly or indirectly from the Android 2.2 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

The Android platform includes a set of managed APIs, a set of native APIs, and a body of so-called "soft" APIs such as the Intent system and web-application APIs. This section details the hard and soft APIs that are integral to compatibility, as well as certain other relevant technical and user interface behaviors. Device implementations MUST comply with all the requirements in this section.

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 2.2 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.

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. This section details the "soft" APIs and system behaviors required for compatibility with Android 2.2. Device implementations MUST meet all the requirements presented in this section.

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
<code>android.os.Build.VERSION.RELEASE</code>	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] .
<code>android.os.Build.VERSION.SDK</code>	The version of the currently-executing Android system, in a format accessible to third-party application code. For Android 2.2, this field MUST have the integer value 8.

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. 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.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. 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.DEVICE	A value chosen by the device implementer identifying the specific configuration or revision of the body (sometimes called "industrial design") of the device. 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.FINGERPRINT	A string that uniquely identifies this build. It SHOULD be reasonably human-readable. It MUST follow this template: \$(BRAND)/\$(PRODUCT)/\$(DEVICE)/\$(BOARD):\$(VERSION.RELEASE)/\$(ID)/\$(VERSION.INCREMENTAL):\$(TYPE)/\$(TAGS) For example: acme/mydevice/generic/generic:2.2/ERC77/3359: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.
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. 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 device. MUST be human-readable, but is not necessarily intended for view by 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.TAGS	A comma-separated list of tags chosen by the device implementer that further distinguish the build. For example, "unsigned,debug". This field MUST NOT be null or the empty string (""), but a single tag (such as "release") is fine.
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".
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

Android uses Intents to achieve loosely-coupled integration between applications. This section describes requirements related to the Intent patterns that **MUST** be honored by device implementations. 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 a phone dialer, calendar, contacts book, 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
- Calculator
- Camera
- Contacts
- Email
- Gallery
- GlobalSearch
- Launcher
- LivePicker (that is, the Live Wallpaper picker application; **MAY** be omitted if the device does not support Live Wallpapers, per Section 3.8.5.)
- Messaging (AKA "Mms")
- Music
- Phone
- Settings
- SoundRecorder

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 component 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 implementers **MUST** allow each Intent pattern referenced in Section 3.2.3.1 to be overridden by third-party applications. The upstream Android open source project 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.

3.2.3.3. Intent Namespaces

Device implementers **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.* 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.

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

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. Device implementations MUST include support for code running in the managed environment to call into native code, using the standard Java Native Interface (JNI) semantics. The following APIs MUST be available to native code:

- libc (C library)
- libm (math library)
- JNI interface
- libz (Zlib compression)
- liblog (Android logging)
- Minimal support for C++
- Support for OpenGL, as described below

Device implementations MUST support OpenGL ES 1.0. Devices that lack hardware acceleration MUST implement OpenGL ES 1.0 using a software renderer. Device implementations SHOULD implement as much of OpenGL ES 1.1 as the device hardware supports. Device implementations SHOULD provide an implementation for OpenGL ES 2.0, if the hardware is capable of reasonable performance on those APIs.

These libraries MUST be source-compatible (i.e. header compatible) and binary-compatible (for a given processor architecture) with the versions provided in Bionic by the Android Open Source project. Since the Bionic implementations are not fully compatible with other implementations such as the GNU C library, device implementers SHOULD use the Android implementation. If device implementers use a different implementation of these libraries, they MUST ensure header, binary, and behavioral compatibility.

Device implementations MUST accurately report the native Application Binary Interface (ABI) supported by the device, via the `android.os.Build.CPU_ABI` API. The ABI MUST be one of the entries documented in the latest version of the Android NDK, in the file `docs/CPU-ARCH-ABIS.txt`. Note that additional releases of the Android NDK may introduce support for additional ABIs.

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

Many developers and applications rely on the behavior of the `android.webkit.WebView` class [\[Resources, 8\]](#) for their user interfaces, so the `WebView` implementation must be compatible across Android implementations. Similarly, a full web experience is central to the Android user experience. Device implementations MUST include a version of `android.webkit.WebView` consistent with the upstream Android software, and MUST include a modern HTML5-capable browser, as described below.

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 533.1 WebKit build from the upstream Android Open Source tree for Android 2.2. 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/533.1 (KHTML, like

Gecko) Version/4.0 Mobile Safari/533.1

- 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`

The WebView configuration MUST include support for the HTML5 database, application cache, and geolocation APIs [\[Resources, 9\]](#). The WebView MUST include support for the HTML5 `<video>` tag. 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 an browser technology other than WebKit. However, even if an alternate Browser application is shipped, 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 [\[Resources, 9\]](#) as possible. Minimally, device implementations MUST support HTML5 geolocation, application cache, and database APIs and the `<video>` tag in standalone the Browser application.

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 meaning 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 particular permission

The above list is not comprehensive, and the onus is on device implementers to ensure behavioral compatibility. 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.

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.

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 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.

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 [\[Resources, 10\]](#).

Device implementations with screens classified as medium- or low-density **MUST** configure Dalvik to allocate at least 16MB of memory to each application. Device implementations with screens classified as high-density **MUST** configure Dalvik to allocate at least 24MB of memory to each application. Note that device implementations **MAY** allocate more memory than these figures.

3.8. User Interface Compatibility

The Android platform includes some developer APIs that allow developers to hook into the system user interface. Device implementations **MUST** incorporate these standard UI APIs into custom user interfaces they develop, as explained below.

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 [\[Resources, 11\]](#). The Android Open Source reference release includes a Launcher application that includes user interface elements allowing the user to add, view, and remove AppWidgets from the home screen.

Device implementers **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 elements 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 implementer **MUST** provide a separate application accessible from the Launcher that allows users to add, configure, view, and remove AppWidgets.

3.8.2. Notifications

Android includes APIs that allow developers to notify users of notable events [\[Resources, 12\]](#). Device implementers **MUST** provide support for each class of notification so defined; specifically: sounds, vibration, light and status bar.

Additionally, the implementation **MUST** correctly render all resources (icons, sound files, etc.) provided for in the APIs [\[Resources, 13\]](#), or in the Status Bar icon style guide [\[Resources, 14\]](#). 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.

3.8.3. Search

Android includes APIs [\[Resources, 15\]](#) 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.

Device implementations MAY ship alternate search user interfaces, but SHOULD include a hard or soft dedicated search button, that can be used at any time within any app to invoke the search framework, with the behavior provided for in the API documentation.

3.8.4. Toasts

Applications can use the "Toast" API (defined in [\[Resources, 16\]](#)) 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. 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 [\[Resources, 17\]](#). 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.

4. Reference Software Compatibility

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

- Calculator (included in SDK)
- Lunar Lander (included in SDK)
- The "Apps for Android" applications [\[Resources, 18\]](#).
- Replica Island (available in Android Market; only required for device implementations that support with OpenGL ES 2.0)

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

Additionally, device implementations MUST test each menu item (including all sub-menus) of each of these smoke-test applications:

- ApiDemos (included in SDK)
- ManualSmokeTests (included in CTS)

Each test case in the applications above MUST run correctly on the device implementation.

5. Application Packaging Compatibility

Device implementations MUST install and run Android ".apk" files as generated by the "aapt" tool included in the official Android SDK [\[Resources, 19\]](#).

Devices implementations MUST NOT extend either the .apk [\[Resources, 20\]](#), Android Manifest [\[Resources, 21\]](#), or Dalvik bytecode [\[Resources, 10\]](#) 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.

6. Multimedia Compatibility

Device implementations MUST fully implement all multimedia APIs. Device implementations MUST include support for all multimedia codecs described below, and SHOULD meet the sound processing guidelines described below.

6.1. Media Codecs

Device implementations MUST support the following multimedia codecs. 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.

	Name	Encoder	Decoder	Details	File/Container Format
Audio	AAC LC/LTP		X	Mono/Stereo content in any combination of standard bit rates up to 160 kbps and sampling rates between 8 to 48kHz	3GPP (.3gp) and MPEG-4 (.mp4, .m4a). No support for raw AAC (.aac)
	HE-AACv1 (AAC+)		X		
	HE-AACv2 (enhanced AAC+)		X		
	AMR-NB	X	X	4.75 to 12.2 kbps sampled @ 8kHz	3GPP (.3gp)
	AMR-WB		X	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16kHz	3GPP (.3gp)
	MP3		X	Mono/Stereo 8-320Kbps constant (CBR) or variable bit-rate (VBR)	MP3 (.mp3)
	MIDI		X	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). Also RTTTL/RTX (.rtttl, .rtx), OTA (.ota), and iMelody (.imy)
	Ogg Vorbis		X		Ogg (.ogg)
	PCM		X	8- and 16-bit linear PCM (rates up to limit of hardware)	WAVE (.wav)
Image	JPEG	X	X	base+progressive	
	GIF		X		
	PNG	X	X		
	BMP		X		
Video	H.263	X	X		3GPP (.3gp) files
	H.264		X		3GPP (.3gp) and MPEG-4 (.mp4) files
	MPEG4 Simple Profile		X		3GPP (.3gp) file

Note that the table above does not list specific bitrate requirements for most video codecs. The reason for this is that in practice, 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.

6.2. Audio Recording

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

- Noise reduction processing, if present, SHOULD be disabled.
- Automatic gain control, if present, SHOULD be disabled.
- 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 5000 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% from 100 Hz to 4000 Hz at 90 dB SPL input level.

Note: while the requirements outlined above are stated as "SHOULD" for Android 2.2, the Compatibility Definition for a future version is planned to change these to "MUST". That is, these requirements are optional in Android 2.2 but **will be required** by a future version. Existing and new devices that run Android 2.2 Android are **very strongly encouraged to meet these requirements in Android 2.2**, or they will not be able to attain Android compatibility when upgraded to the future version.

6.3. Audio Latency

Audio latency is broadly defined as the interval between when an application requests an audio playback or record operation, and when the device implementation actually begins the operation. Many classes of applications rely on short latencies, to achieve real-time effects such sound effects or VOIP communication. Device implementations SHOULD meet all audio latency requirements outlined in this section.

For the purposes of this section:

- "cold output latency" is defined to be the interval between when an application requests audio playback and when sound begins playing, when the audio system has been idle and powered down prior to the request
- "warm output latency" is defined to be the interval between when an application requests audio playback and when sound begins playing, when the audio system has been recently used but is currently idle (that is, silent)
- "continuous output latency" is defined to be the interval between when an application issues a sample to be played and when the speaker physically plays the corresponding sound, while the device is currently playing back audio
- "cold input latency" is defined to be the interval between when an application requests audio recording and when the first sample is delivered to the application via its callback, when the audio system and microphone has been idle and powered down prior to the request
- "continuous input latency" is defined to be when an ambient sound occurs and when the sample corresponding to that sound is delivered to a recording application via its callback, while the device is in recording mode

Using the above definitions, device implementations SHOULD exhibit each of these properties:

- cold output latency of 100 milliseconds or less
- warm output latency of 10 milliseconds or less
- continuous output latency of 45 milliseconds or less
- cold input latency of 100 milliseconds or less
- continuous input latency of 50 milliseconds or less

Note: while the requirements outlined above are stated as "SHOULD" for Android 2.2, the Compatibility Definition for a future version is planned to change these to "MUST". That is, these requirements are optional in Android 2.2 but **will be required** by a future version. Existing and new devices that run Android 2.2 Android are **very strongly encouraged to meet these requirements in Android 2.2**, or they will not be able to attain Android compatibility when upgraded to the future version.

7. Developer Tool Compatibility

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)** [\[Resources, 19\]](#)
Device implementations MUST support all adb functions as documented in the Android SDK. The device-side adb daemon SHOULD be inactive by default, but there MUST be a user-accessible mechanism to turn on the Android Debug Bridge.
- **Dalvik Debug Monitor Service (known as ddms)** [\[Resources, 19\]](#)
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** [\[Resources, 22\]](#)
Device implementations MUST include the Monkey framework, and make it available for applications to use.

8. Hardware Compatibility

Android is intended to support device implementers creating innovative form factors and configurations. At the same time Android developers expect certain hardware, sensors and APIs across all Android device. This section lists the hardware features that all Android 2.2 compatible devices must support.

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 defined 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:

- class definitions for the component's APIs MUST 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

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. [\[Resources, 23\]](#)

8.1. Display

Android 2.2 includes facilities that perform certain automatic scaling and transformation operations under some circumstances, to ensure that third-party applications run reasonably well on a variety of hardware configurations [\[Resources, 24\]](#). Devices MUST properly implement these behaviors, as detailed in this section.

For Android 2.2, these are the most common display configurations:

Screen Type	Width (Pixels)	Height (Pixels)	Diagonal Length Range (inches)	Screen Size Group	Screen Density Group
QVGA	240	320	2.6 - 3.0	Small	Low
WQVGA	240	400	3.2 - 3.5	Normal	Low
FWQVGA	240	432	3.5 - 3.8	Normal	Low
HVGA	320	480	3.0 - 3.5	Normal	Medium
WVGA	480	800	3.3 - 4.0	Normal	High
FWVGA	480	854	3.5 - 4.0	Normal	High
WVGA	480	800	4.8 - 5.5	Large	Medium
FWVGA	480	854	5.0 - 5.8	Large	Medium

Device implementations corresponding to one of the standard configurations above MUST be configured to report the indicated screen size to applications via the `android.content.res.Configuration` [\[Resources, 24\]](#) class.

Some .apk packages have manifests that do not identify them as supporting a specific density range. When running such applications, the following constraints apply:

- Device implementations MUST interpret resources in a .apk that lack a density qualifier as defaulting to "medium" (known as "mdpi" in the SDK documentation.)
- When operating on a "low" density screen, device implementations MUST scale down medium/mdpi assets by a factor of 0.75.
- When operating on a "high" density screen, device implementations MUST scale up medium/mdpi assets by a factor of 1.5.
- Device implementations MUST NOT scale assets within a density range, and MUST scale assets by exactly these factors between density ranges.

8.1.2. Non-Standard Display Configurations

Display configurations that do not match one of the standard configurations listed in Section 8.1.1 require additional consideration and work to be compatible. Device implementers MUST contact Android Compatibility Team as described in Section 13 to obtain classifications for screen-size bucket, density, and scaling factor. When provided with this information, device implementations MUST implement them as specified.

Note that some display configurations (such as very large or very small screens, and some aspect ratios) are fundamentally incompatible with Android 2.2; therefore device implementers are encouraged to contact Android Compatibility Team as early as possible in the development process.

8.1.3. Display Metrics

Device implementations MUST report correct values for all display metrics defined in `android.util.DisplayMetrics` [\[Resources, 26\]](#).

8.1.4. Declared Screen Support

Applications may 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, medium, and large screens, as described in the Android SDK documentation.

8.2. 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 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` [\[Resources, 25\]](#) (that is, QWERTY, or 12-key)

8.3. Non-touch Navigation

Device implementations:

- MAY omit a non-touch navigation options (that is, may omit a trackball, d-pad, or wheel)
- MUST report the correct value for `android.content.res.Configuration.navigation` [\[Resources, 25\]](#)

8.4. Screen Orientation

Compatible 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.

8.5. Touchscreen input

Device implementations:

- MUST have a touchscreen
- MAY have either capacitive or resistive touchscreen
- MUST report the value of `android.content.res.Configuration` [\[Resources, 25\]](#) reflecting corresponding to the type of the specific touchscreen on the device
- SHOULD support fully independently tracked pointers, if the touchscreen supports multiple pointers

8.6. USB

Device implementations:

- MUST implement a USB client, connectable to a USB host with a standard USB-A port
- MUST implement the Android Debug Bridge over USB (as described in Section 7)
- MUST implement the USB mass storage specification, to allow a host connected to the device to access the contents of the `/sdcard` volume
- SHOULD use the micro USB form factor on the device side
- MAY include a non-standard port on the device side, but if so MUST ship with a cable capable of connecting the custom pinout to standard USB-A port
- SHOULD implement support for the USB Mass Storage specification (so that either removable or fixed storage on the device can be accessed from a host PC)

8.7. 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, regardless of application state. These functions SHOULD be implemented via dedicated buttons. They MAY be implemented using software, gestures, touch panel, etc., but if so they MUST be always accessible and not obscure or interfere with the available application display area.

Device implementers SHOULD also provide a dedicated search key. Device implementers MAY also provide send and end keys for phone calls.

8.8. Wireless Data Networking

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

If a device implementation includes a particular modality for which the Android SDK includes an API (that is, WiFi, GSM, or CDMA), the implementation MUST support the API.

Devices MAY implement more than one form of wireless data connectivity. Devices MAY implement wired data connectivity (such as Ethernet), but MUST nonetheless include at least one form of wireless connectivity, as above.

8.9. Camera

Device implementations MUST include a rear-facing camera. The included rear-facing camera:

- 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`.

Device implementations MUST implement the following behaviors for the camera-related APIs:

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. (This is the format used natively by the 7k hardware family.) That is, NV21 MUST be the default.

Device implementations MUST implement the full Camera API included in the Android 2.2 SDK documentation ([\[Resources, 27\]](#)), 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.)

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.

Device implementations MAY include a front-facing camera. However, if a device implementation includes a front-facing camera, the camera API as implemented on the device MUST NOT use the front-facing camera by default. That is, the camera API in Android 2.2 is for rear-facing cameras only, and device implementations MUST NOT reuse or overload the API to act on a front-facing camera, if one is present. Note that any custom APIs added by device implementers to support front-facing cameras MUST abide by sections 3.5 and 3.6; for instance, if a custom `android.hardware.Camera` or `Camera.Parameters` subclass is provided to support front-facing cameras, it MUST NOT be located in an existing namespace, as described by sections 3.5 and 3.6. Note that the inclusion of a front-facing camera does not meet the requirement that devices include a rear-facing camera.

8.10. Accelerometer

Device implementations MUST include a 3-axis accelerometer and MUST be able to deliver events at 50 Hz or greater. The coordinate system used by the accelerometer MUST comply with the Android sensor coordinate system as detailed in the Android APIs (see [\[Resources, 28\]](#)).

8.11. Compass

Device implementations MUST include a 3-axis compass and MUST be able to deliver events 10 Hz or greater. The coordinate system used by the compass MUST comply with the Android sensor coordinate system as defined in the Android API (see [\[Resources, 28\]](#)).

8.12. GPS

Device implementations MUST include a GPS receiver, and SHOULD include some form of "assisted GPS" technique to minimize GPS lock-on time.

8.13. Telephony

Android 2.2 MAY be used on devices that do not include telephony hardware. That is, Android 2.2 is compatible with devices that are not phones. However, if a device implementation does include GSM or CDMA telephony, it MUST implement the full support for the API for that technology. Device implementations that do not include telephony hardware MUST implement the full APIs as no-ops.

See also Section 8.8, Wireless Data Networking.

8.14. Memory and Storage

Device implementations **MUST** have at least 92MB of memory available to the kernel and userspace. The 92MB **MUST** be in addition to any memory dedicated to hardware components such as radio, memory, and so on that is not under the kernel's control.

Device implementations **MUST** have at least 150MB of non-volatile storage available for user data. That is, the `/data` partition **MUST** be at least 150MB.

Beyond the requirements above, device implementations **SHOULD** have at least 128MB of memory available to kernel and userspace, in addition to any memory dedicated to hardware components that is not under the kernel's control. Device implementations **SHOULD** have at least 1GB of non-volatile storage available for user data. Note that these higher requirements are planned to become hard minimums in a future version of Android. Device implementations are strongly encouraged to meet these requirements now, or else they may not be eligible for compatibility for a future version of Android.

8.15. Application Shared Storage

Device implementations **MUST** offer shared storage for applications. The shared storage provided **MUST** be at least 2GB 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, the shared storage **MUST** implement USB mass storage, as described in Section 8.6. As shipped out of the box, the shared storage **MUST** be mounted with the FAT filesystem.

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 2GB 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 2GB in size or larger, formatted as FAT, 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.

8.16. Bluetooth

Device implementations **MUST** include a Bluetooth transceiver. Device implementations **MUST** enable the RFCOMM-based Bluetooth API as described in the SDK documentation [[Resources, 30](#)]. 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.

9. Performance Compatibility

One of the goals of the Android Compatibility Program is to enable consistent application experience to consumers. Compatible implementations must ensure not only that applications simply run correctly on the device, but that they do so with reasonable performance and overall good user experience. Device implementations **MUST** meet the key performance metrics of an Android 2.2 compatible device defined in the table below:

Metric	Performance Threshold	Comments
--------	-----------------------	----------

Application Launch Time	<p>The following applications should launch within the specified time.</p> <ul style="list-style-type: none"> • Browser: less than 1300ms • MMS/SMS: less than 700ms • AlarmClock: less than 650ms 	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.	

10. 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 [\[Resources, 29\]](#) 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.

10.1. Permissions

Device implementations MUST support the Android permissions model as defined in the Android developer documentation [\[Resources, 29\]](#). 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.

10.2. 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 [\[Resources, 29\]](#).

10.3. Filesystem Permissions

Device implementations MUST support the Android file access permissions model as defined in as defined in the Security and Permissions reference [\[Resources, 29\]](#).

10.4. 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 10.

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.

11. 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 2.2. Device implementations **MUST** pass the latest CTS version available at the time the device software is completed.

12. 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. 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.

13. 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-driven 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.