Getting Started
Compatibility Overview
Current CDD
CTS Introduction
CTS Development

More Information
Downloads
FAQs
Contact Us

Android Compatibility

Android's purpose is to establish an open platform for developers to build innovative mobile apps. Three key components work together to realize this platform.

The Android Compatibility Program defines the technical details of Android platform and provides tools used by OEMs to ensure that developers' apps run on a variety of devices. The Android SDK provides built-in tools that Developers use to clearly state the device features their apps require. And Android Market shows apps only to those devices that can properly run them.

These pages describe the Android Compatibility Program and how to get access to compatibility information and tools. The latest version of the Android source code and compatibility program is 2.3, which corresponded to the Gingerbread branch.

Why build compatible Android devices?

Users want a customizable device.

A mobile phone is a highly personal, always-on, always-present gateway to the Internet. We haven't met a user yet who didn't want to customize it by extending its functionality. That's why Android was designed as a robust platform for running after-market applications.

Developers outnumber us all.

No device manufacturer can hope to write all the software that a person could conceivably need. We need third-party developers to write the apps users want, so the Android Open Source Project aims to make it as easy and open as possible for developers to build apps.

Everyone needs a common ecosystem.

Every line of code developers write to work around a particular phone's bug is a line of code that didn't add a new feature. The more compatible phones there are, the more apps there will be. By building a fully compatible Android device, you benefit from the huge pool of apps written for Android, while increasing the incentive for developers to build more of those apps.

Android compatibility is free, and it's easy.

If you are building a mobile device, you can follow these steps to make sure your device is compatible with Android. For more details about the Android compatibility program in general, see the program overview.

Building a compatible device is a three-step process:

1. *Obtain the Android software source code*. This is the source code for the Android platform, that you port to your hardware.
2. *Comply with Android Compatibility Definition Document (CDD)*. The CDD enumerates the software and

hardware requirements of a compatible Android device.

3. *Pass the Compatibility Test Suite (CTS)*. You can use the CTS (included in the Android source code) as an ongoing aid to compatibility during the development process.

Joining the Ecosystem

Once you've built a compatible device, you may wish to include Android Market to provide your users access to the third-party app ecosystem. Unfortunately, for a variety of legal and business reasons, we aren't able to automatically license Android Market to all compatible devices. To inquire about access about Android Market, you can contact us.

Compatibility Program Overview

The Android compatibility program makes it easy for mobile device manufacturers to develop compatible Android devices.

Program goals

The Android compatibility program works for the benefit of the entire Android community, including users, developers, and device manufacturers.

Each group depends on the others. Users want a wide selection of devices and great apps; great apps come from developers motivated by a large market for their apps with many devices in users' hands; device manufacturers rely on a wide variety of great apps to increase their products' value for consumers.

Our goals were designed to benefit each of these groups:

- *Provide a consistent application and hardware environment to application developers.* Without a strong compatibility standard, devices can vary so greatly that developers must design different versions of their applications for different devices. The compatibility program provides a precise definition of what developers can expect from a compatible device in terms of APIs and capabilities. Developers can use this information to make good design decisions, and be confident that their apps will run well on any compatible device.
- *Enable a consistent application experience for consumers.* If an application runs well on one compatible Android device, it should run well on any other device that is compatible with the same Android platform version. Android devices will differ in hardware and software capabilities, so the compatibility program also provides the tools needed for distribution systems such as Android Market to implement appropriate filtering. This means that users can only see applications which they can actually run.
- *Enable device manufacturers to differentiate while being compatible.* The Android compatibility program focuses on the aspects of Android relevant to running third-party applications, which allows device manufacturers the flexibility to create unique devices that are nonetheless compatible.
- *Minimize costs and overhead associated with compatibility.* Ensuring compatibility should be easy and inexpensive to device manufacturers. The testing tool (CTS) is free, open source, and available for download. CTS is designed to be used for continuous self-testing during the device development process to eliminate the cost of changing your workflow or sending your device to a third party for testing. Meanwhile, there are no required certifications, and thus no corresponding costs and fees.

The Android compatibility program consists of three key components:

- The source code to the Android software stack
- The Compatilbility Definition Document, representing the "policy" aspect of compatibility
- The Compatilbility Test Suite, representing the "mechanism" of compatibility

Just as each version of the Android platform exists in a separate branch in the source code tree, there is a separate CTS and CDD for each version as well. The CDD, CTS, and source code are -- along with your hardware and your software customizations -- everything you need to create a compatible device.

Compatibility Definition Document (CDD)

For each release of the Android platform, a detailed Compatibility Definition Document (CDD) will be provided. The CDD represents the "policy" aspect of Android compatibility.

No test suite, including CTS, can truly be comprehensive. For instance, the CTS includes a test that checks for the presence and correct behavior of OpenGL graphics APIs, but no software test can verify that the graphics actually appear correctly on the screen. More generally, it's impossible to test the presence of hardware features such as keyboards, display density, WiFi, and Bluetooth.

The CDD's role is to codify and clarify specific requirements, and eliminate ambiguity. The CDD does not attempt to be comprehensive. Since Android is a single corpus of open-source code, the code itself is the comprehensive "specification" of the platform and its APIs. The CDD acts as a "hub", referencing other content (such as SDK API documentation) that provides a framework in which the Android source code may be used so that the end result is a compatible system.

If you want to build a device compatible with a given Android version, start by checking out the source code for that version, and then read the corresponding CDD and stay within its guidelines. For additional details, simply examine the latest CDD.

Compatibility Test Suite (CTS)

The CTS is a free, commercial-grade test suite, available for download. The CTS represents the "mechanism" of compatibility.

The CTS runs on a desktop machine and executes test cases directly on attached devices or an emulator. The CTS is a set of unit tests designed to be integrated into the daily workflow (such as via a continuous build system) of the engineers building a device. Its intent is to reveal incompatibilities early on, and ensure that the software remains compatible throughout the development process.

For details on the CTS, consult the CTS introduction.

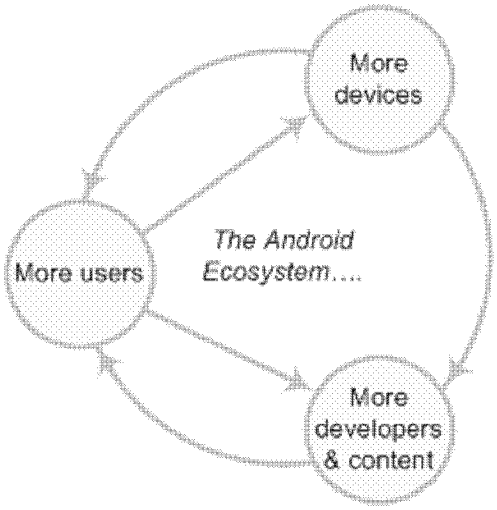
android compatibility program

Compatibility Test Suite (CTS) Framework User Manual

Android 1.6 CTS r4
Open Handset Alliance

Contents

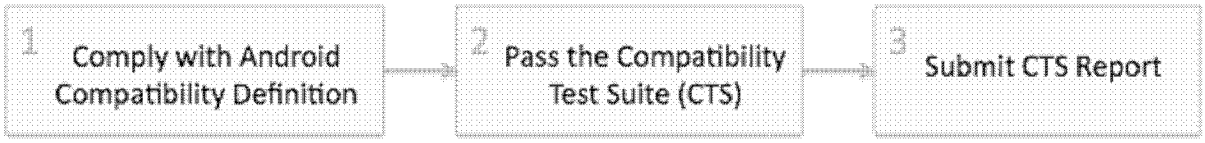
1. Why be compatible?



1. *Give your users the best possible experience with the applications they run.*
   When a device is compatible with Android, users can choose from among many high-quality applications. Applications that take full advantage of Android's features are likely to perform best on compatible devices.
2. *Make it easy for developers to write top-quality applications for your device.*
   Developers want to streamline their applications for Android, and this is easiest for them when they are writing for a predictable platform.
3. *Take advantage of the Android Market.*
   Compatible handsets can give users access to the Android Market.

Android compatibility is free, and it's easy.

2. How can I become compatible?



2.1. Comply with Android Compatibility Definition document

To start, read the Android compatibility definition for the Android platform version that you want. This document enumerates the software and the hardware features in a compatible Android device. Except where noted, the features are all required for Android compliance. To learn more about Android compatibility definition in general, and to locate and download a particular definitions document, see the current Compatibility Definition. Archived versions of older Compatibility Definitions may be found on the Downloads page.

2.2. Pass the Compatibility Test Suite (CTS)

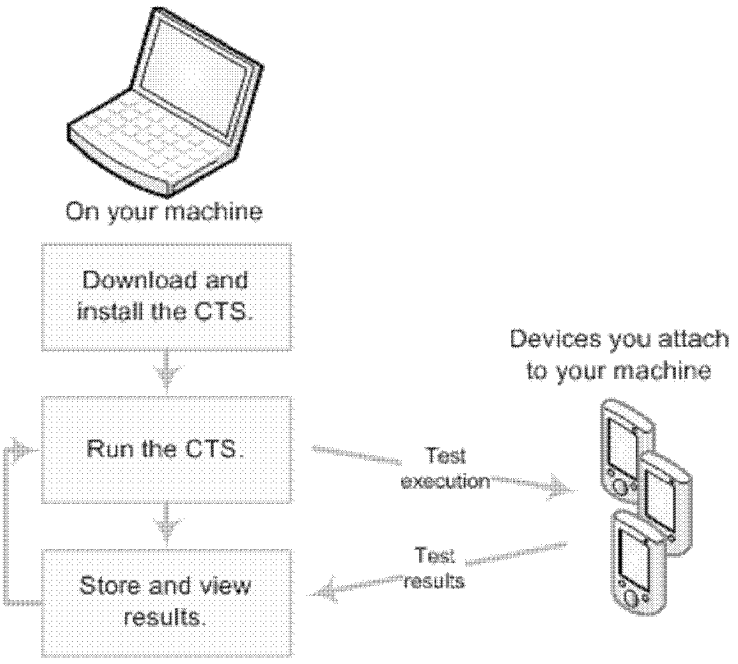
The Compatibility Test Suite (CTS) is a downloadable open-source testing harness that you can use in any way you like as you develop your handset; for example, you could use the CTS to do continuous self-testing during your development work. For more about the CTS and the compatibility report that it generates, see the Compatibility Test Suite page. For instructions on using the CTS, see the CTS User Guide.

2.3. Submit report

When you are ready to claim compatibility for your device, you can submit the CTS-generated report to cts@android.com. When you submit a CTS report, you can also request access to the Android Market.

* This is an early preview of CTS. The compatibility site and the service to certify your compatibility reports are work in progress - we will update you when these are ready.

3. How does the CTS work?



The CTS is an automated testing harness that includes two major software components:

- The CTS test harness runs on your desktop machine and manages test execution.
- Individual test cases are executed on attached mobile devices or on an emulator. The test cases are written in Java as JUnit tests and packaged as Android .apk files to run on the actual device target.

3.1. Workflow

1. Use the bundled CTS release or download the CTS from the Android Open Source Project onto your desktop machine.
2. Install and configure the CTS.
3. Attach at least one device (or emulator) to your machine.
4. Launch the CTS. The CTS test harness loads the test plan onto the attached devices. For each test in the test harness:
    - The test harness pushes a .apk file to each device, executes the test through instrumentation, and records test results.
    - The test harness removes the .apk file from each device.
5. Once all the tests are executed, you can view the test results in your browser and use the results to adjust your design. You can continue to run the CTS throughout your development process.

When you are ready, you can submit the report generated by the CTS to cts@android.com. The report is a .zip archived file that contains XML results and supplemental information such as screen captures.

3.2. Types of test cases

The CTS includes the following types of test cases:

- *Unit tests* test atomic units of code within the Android platform; e.g. a single class, such as java.util.HashMap.
- *Functional tests* test a combination of APIs together in a higher-level use-case.
- *Reference application tests* instrument a complete sample application to exercise a full set of APIs and Android runtime services

Future versions of the CTS will include the following types of test cases:

- *Robustness tests* test the durability of the system under stress.
- *Performance tests* test the performance of the system against defined benchmarks, for example rendering frames per second.

3.3. Areas Covered

The unit test cases cover the following areas to ensure compatibility

Area	Description
Signature tests	For each Android release, there are XML files describing the signatures of all public APIs contained in the release. The CTS contains a utility to check those API signatures against the APIs available on the device. The results from signature checking are recorded in the test result XML file.
Platform API Tests	Test the platform (core libraries and Android Application Framework) APIs as documented in the SDK Class Index to ensure API correctness: • correct class, attribute and method signatures • correct method behavior • negative tests to ensure expected behavior for incorrect parameter handling
Dalvik VM Tests	The tests focus on testing the Dalvik VM

Platform Data Model	The CTS tests the core platform data model as exposed to application developers through content providers, as documented in the SDK android.provider package: • contacts • browser • settings • more...
Platform Intents	The CTS tests the core platform intents, as documented in the SDK Available Intents.
Platform Permissions	The CTS tests the core platform permissions, as documented in the SDK Available Permissions.
Platform Resources	The CTS tests for correct handling of the core platform resource types, as documented in the SDK Available Resource Types. This includes tests for: • simple values • drawables • nine-patch • animations • layouts • styles and themes • loading alternate resources

4. Setting up and using the CTS

4.1. Configuring the CTS

To run CTS, make sure you have atleast the Android 1.6 r1 SDK installed on your machine. ****There are changes to adb in 1.6 that will cause CTS to not work correctly with older versions of adb.****

To configure CTS, extract the contents of the zip file and edit the `android-cts/tools/startcts` script - modify the variable `SDK_ROOT` to match your environment.

Example:

```
SDK_ROOT=/home/myuser/android-sdk-linux_x86-1.6_r1
```

This should point to the top-level directory where you unzipped the Android 1.6 SDK to.

4.2. Setting up your device

CTS can be executed only on consumer device since Android 1.6 -- you can run CTS only on developer builds for Android 1.0 and 1.5.

This section is important as not following these instructions will lead to test timeouts/ failures:

1. Please download and install the Android 1.6 SDK on your machine.
2. Your phone should be running a **user build (Android 1.6 and later)** from source.android.com
3. Please refer to this link on the Android developer site and set up your device accordingly.
4. Make sure that your device has been flashed with a user build (Android 1.6 and later) before you run CTS.
5. You need to download the TTS files via Settings > Speech synthesis > Install voice data before running CTS tests. (Note that this assumes you have Android Market installed on the device, if not you will need to install the files manually via adb)
6. It is advisable to log in to the device with a test Google account, not an account that you actually use.
7. Make sure the device has a SD card plugged in and the card is empty. *Warning: CTS may modify/erase data on the SD card plugged in to the device.*
8. Do a factory data reset on the device (Settings > SD Card & phone storage > Factory data reset). *Warning: This will erase all user data from the phone.*
9. Make sure no lock pattern is set on the device (Settings > Security & location > Require Pattern should be unchecked.

10. Make sure the "Screen Timeout" is set to "Never Timeout" (Settings > Sound & Display > Screen Timeout should be set to "Never Timeout".
11. Make sure the "Stay Awake" development option is checked (Settings > Applications > Development > Stay awake).
12. Make sure Settings > Application > Development > Allow mock locations is set to true.
13. Make sure the device is at the home screen at the start of CTS (Press the home button).
14. While a device is running tests, it must not be used for any other tasks.
15. Do not press any keys on the device while CTS is running. Pressing keys or touching the screen of a test device will interfere with the running tests and may lead to test failures.

4.3. Using the CTS

To run a test plan:

1. Make sure you have at least one device connected (or the emulator running). Launch the CTS console by running the *startcts* script which you modified to match your environment, e.g.
   `$ bash android-cts/tools/startcts`
2. You may start the default test plan (containing all of the test packages) by typing `start --plan CTS`. This will kick off all the CTS tests required for compatibility.
   Type `ls -p` to see a list of test packages in the repository.
   Type `ls --plan` to see a list of test plans in the repository.
   See the CTS command reference or type `help` for a complete list of supported commands.
3. Alternately, you can just run a CTS plan from the command line using
   `startcts start --plan <plan_name>`
4. You should test progress and results reported on the console.

4.4. Selecting CTS Plans

For this release the following 7 test plans are available.

1. `CTS` - contains all tests and will run ~21,000 tests on your device. These tests are required for compatibility. At this point performance tests are not part of this plan (this will change for future CTS releases).
2. `Signature` - contains the signature verification of all public APIs
3. `Android` - contains tests for the android APIs
4. `Java` - contains tests for the Java core library
5. `VM` - contains tests for the Dalvik virtual machine

6. `RefApp` - contains reference application tests (more coming in future CTS release)
7. `Performance` - contains performance tests for your implementation (more coming in future CTS releases)

These can be executed with the `start` command as mentioned earlier.

5. Interpreting the Test Results

The test results are placed in the file:
`$CTS_ROOT/repository/results/<start time>.zip`
Inside the zip, the `testResult.xml` file contains the actual results -- open this file in any web browser (Firefox 3.x recommended) to view the test results.

android compatibility program **Test Report for dream - HT851LZ01986**

Device Information		Test Summary	
Device Make	dream		
Build model	HT851LZ01986		
Firmware Version	1.5		
Firmware Build Number	CUPCAKE	Plan name	CTS
Android Platform Version	3	Start time	Wed Feb 11 15:20:53 PST 2009
Supported Locales	en_US;es;en_US;zz_ZZ;en;	End time	Wed Feb 11 15:49:49 PST 2009
Screen size	320x480	Version	1.0
Phone number	null		
x dpi	180.62193		
y dpi	181.96814	Tests Passed	1448
Touch	finger	Tests Failed	40
Navigation	trackball	Tests Timed out	1
Keypad	qwerty	Tests Not Executed	0
Network			
IMEI	351676030149928		
IMSI	null		

Test Summary by Package

Test Package	Tests Passed
android.tests.sigtest	1 / 1
android.app	32 / 35
android.content	163 / 167
android.database	18 / 18
android.graphics	489 / 499
android.location	19 / 19
android.net	29 / 30
android.os	75 / 75
android.provider	10 / 10
android.text	147 / 150
android.util	32 / 32

The *'device information'* section provides details about the device and the firmware (make, model, firmware build, platform) and the hardware on the device (screen resolution, keypad, screen type).
The details of the executed test plan are present in the *'test summary'* section which provides the CTS plan name and execution start and end times. It also presents an aggregate summary of the number of tests that passed, failed, time out or could not be executed.
The next section also provides a summary of tests passed per package.

Compatibility Test Package: android.widget

Test	Result	Failure Details
Suite: android.widget.cts.		
- AbsSeekBarTest		
-- testConstructor	pass	
-- testAccessThumbOffset	pass	
-- testSetThumb	pass	
-- testOnTouchEvent	pass	
-- testDrawableStateChanged	pass	
-- testOnDraw	pass	
-- testOnMeasure	pass	
-- testVerifyDrawable	fail	junit.framework.AssertionFailedError at android.widget.cts.AbsSeekBarTest.testVerifyDrawable(AbsSeekBarTest.java:327)
-- testOnSizeChanged	pass	
-- testAndroidTestCaseSetupProperly	pass	
- ButtonTest		
-- testConstructor	pass	
-- testAndroidTestCaseSetupProperly	pass	
- ChronometerTest		
-- testConstructor	pass	
-- testAccessBase	pass	
-- testAccessFormat	pass	
-- testOnDetachedFromWindow	pass	
-- testOnWindowVisibilityChanged	pass	
-- testStartAndStop	pass	
- CompoundButtonTest		
-- testConstructor	pass	
-- testAccessChecked	pass	
-- testSetOnCheckedChangeListener	pass	
-- testToggle	pass	
-- testPerformClick	pass	
-- testDrawableStateChanged	pass	
-- testSetButtonDrawableByDrawable	pass	
-- testSetButtonDrawableById	pass	
-- testOnCreateDrawableState	pass	
-- testOnDraw	pass	
-- testAccessInstanceState	pass	

This is followed by details of the the actual tests that were executed. The report lists the test package, test suite, test case and the executed tests. It shows the result of the test execution - pass, fail, timed out or not executed. In the event of a test failure details are provided to help diagnose the cause. Further, the stack trace of the failure is available in the XML file but is not included in the report to ensure brevity - viewing the XML file with a text editor should provide details of the test failure (search for the <*Test*> tag corresponding to the failed test and look within it for the <*StackTrace*> tag).

6. Release Notes

6.1. General

- This CTS release contains approximately 21,000 tests that you can execute on the device.
- Please make sure all steps in section 4.2 "Setting up your device" have been followed before you kick off CTS. Not following these instructions may cause tests to timeout or fail.

6.2. Known Issues

- The framework restarts the device periodically -- this is expected behavior.
- Concurrent devices are not supported in this release -- CTS can be executed on only one device at a given time.
- The CTS console allows the user to derive a new test plan based on previous results. This is useful for re-running tests that did not pass in a previous run. Successive derivation of test plans (i.e. deriving a test plan from test results of an already derived test plan) may result in the plan including extra tests -- this is a known issue for this release.
- Occasionally while running the tests, a system dialog may pop up informing the user that process 'android.process.acore' is not responding. The user is given the option to kill the process or wait for it to respond. This alert dialog interferes with some tests by grabbing all key and pointer events, causing the tests to fail. Re-running the tests usually fixes the problem.

7. Appendix: CTS Console Command Reference

Host	
help	Display the list of available commands.
exit	Exit the CTS console.
Test Plan	
ls --plan [<test_plan_name>]	Displays the contents of the specified test plan. If no plan is specified, a list of all plans is displayed.
add --plan <new_plan_name>	Create a new test plan. The console will guide you through the test packages to select the tests you want to include in your plan. Note that the plan name must be unique.
add --derivedplan <new_plan_name> [-s <session_id>] [-r [pass | fail | timeout | notExecuted]]	Create a new test plan from an existing result. This test plan will consist of all test with the specified result type in the specified session. If no result type is given, all but the passed tests are included. If no session is given, the latest results are used.
rm --plan <test_plan_name | all>	Remove the specified plan from the plan repository. *all* removes all test plans
start --plan <test_plan_name>	Start the specified test plan and displays progress information. The console will only prompt for further commands when the plan has run to completion. If there are available test sessions for the specified test plan, the CTS console will prompt user to choose between two options: (1) Choose a session from the existing sessions;

	(2) Create a new session. If more than one device connected, CTS host will prompt user to choose one device.
-d, --device <device_id>	Start the specified test plan using the specified device.
-t, --test <test_name>	Start to run the specified test contained in the specified test plan
-p, --package <java_package_name>	Start to run the specified Java package contained in the specified plan.
Test Package	
ls *-p, --package [<package name>]*	List all available test packages in the repository. If package name is specified then lists all its test suites/test cases.
add -p, --package <zip_file_path>	Add new packages to the case repository.
rm -p, --package [<package_name> | all]	Remove the specified package from the case repository. *all* removes all packages
Test Result	
ls *-r, --result* *[pass | fail | timeout | notExecuted]* *[-s <session_id>]*	List the results for all available sessions. If session_id is specified, then lists results for that specific session. If pass, fail, etc. is specified then filters test results based on the specified results.
History	
history | h [<count>] [-e <num>]	List all commands in history. If count is specified, last count commands in history are shown -e allows the command with number num to be executed directly from history
Device	
ls -d, --device	List all attached devices.

android compatibility program

Android Compatibility Definition: Android 1.6

Android 1.6 r2
Google Inc.
compatibility@android.com

Table of Contents

1. Introduction

This document enumerates the requirements that must be met in order for mobile phones to be compatible with Android 1.6. This definition assumes familiarity with the Android Compatibility Program [Resources, 1].

The use of "must", "must not", "required", "shall", "shall not", "should", "should not", "recommended", "may" and "optional" is per the IETF standard defined in RFC2119 [Resources, 2].

As used in this document, a "device implementer" or "implementer" is a person or organization developing a hardware/software solution running Android 1.6. A "device implementation" or "implementation" is the hardware/software solution so developed.

To be considered compatible with Android 1.6, device implementations:

1. MUST meet the requirements presented in this Compatibility Definition, including any documents incorporated via reference.
2. MUST pass the Android Compatibility Test Suite (CTS) available as part of the Android Open Source Project [Resources, 3]. The CTS tests most, **but not all**, 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, 4] 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.

2. Resources

This Compatibility Definition makes reference to a number of resources that can be obtained here.

1. Android Compatibility Program Overview: https://sites.google.com/a/android.com/compatibility/how-it-works
2. IETF RFC2119 Requirement Levels: http://www.ietf.org/rfc/rfc2119.txt
3. Compatibility Test Suite: http://sites.google.com/a/android.com/compatibility/compatibility-test-suite--cts
4. Android Open Source Project: http://source.android.com/
5. API definitions and documentation: http://developer.android.com/reference/packages.html
6. Content Providers: http://code.google.com/android/reference/android/provider/package-summary.html
7. Available Resources: http://code.google.com/android/reference/available-resources.html
8. Android Manifest files: http://code.google.com/android/devel/bblocks-manifest.html
9. Android Permissions reference: http://developer.android.com/reference/android/Manifest.permission.html
10. Build Constants: http://developer.android.com/reference/android/os/Build.html
11. WebView: http://developer.android.com/reference/android/webkit/WebView.html
12. Gears Browser Extensions: http://code.google.com/apis/gears/

13. Dalvik Virtual Machine specification, found in the dalvik/docs directory of a source code checkout; also available at http://android.git.kernel.org/?p=platform/dalvik.git;a=tree;f=docs;h=3e2ddbcaf7f370246246f9f03620a7caccbfcb12;hb=HEAD
14. AppWidgets: http://developer.android.com/guide/practices/ui_guidelines/widget_design.html
15. Notifications: http://developer.android.com/guide/topics/ui/notifiers/notifications.html
16. Status Bar icon style guide: http://developer.android.com/guide/practices/ui_guideline/icon_design.html#statusbarstructure
17. Search Manager: http://developer.android.com/reference/android/app/SearchManager.html
18. Toast: http://developer.android.com/reference/android/widget/Toast.html
19. Apps For Android: http://code.google.com/p/apps-for-android
20. Android apk file description: http://developer.android.com/guide/topics/fundamentals.html
21. Android Debug Bridge (adb): http://code.google.com/android/reference/adb.html
22. Dalvik Debug Monitor Service (ddms): http://code.google.com/android/reference/ddms.html
23. Monkey: http://developer.android.com/guide/developing/tools/monkey.html
24. Display-Independence Documentation:
25. Configuration Constants: http://developer.android.com/reference/android/content/res/Configuration.html
26. Display Metrics: http://developer.android.com/reference/android/util/DisplayMetrics.html
27. 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

Many of these resources are derived directly or indirectly from the Android 1.6 SDK, and will be functionally identical to the information in that SDK's documentation. In any cases where this Compatibility Definition 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 both a set of managed ("hard") APIs, and a body of so-called "soft" APIs such as the Intent system, native-code APIs, 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 1.6 SDK, such as:

1. Core Android Java-language APIs [Resources, 5].
2. Content Providers [Resources, 6].
3. Resources [Resources, 7].
4. AndroidManifest.xml attributes and elements [Resources, 8].

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 1.6. 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, 9]. Note that Section 10 lists addtional 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, 10] 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. For Android 1.6, this field MUST have the string value "1.6".
android.os.Build.VERSION.SDK	The version of the currently-executing Android system, in a format accessible to third-party application code. For Android 1.6, this field MUST have the integer value 4.
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 shipped 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: $(PRODUCT_BRAND)/$(PRODUCT_NAME)/$(PRODUCT_DEVICE)/ $(TARGET_BOOTLOADER_BOARD_NAME):$(PLATFORM_VERSION)/ $(BUILD_ID)/$(BUILD_NUMBER):$(TARGET_BUILD_VARIANT)/ $(BUILD_VERSION_TAGS) For example: acme/mydevicel/generic/generic:Donut/ERC77/ 3359:userdebug/test-keys The fingerprint MUST NOT include spaces. If other fields included in the template above have spaces, they SHOULD be replaced with the ASCII underscore ("_") character in the fingerprint.
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 by the same as android.os.Build.VERSION.INCREMENTAL, but SHOULD be a value intended to be somewhat meaningful for 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.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, Service, or other component 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.) Device implementions MUST support all Intent patterns listed in Appendix A.

3.2.3.2. Intent Overrides

As Android is an extensible platform, device implementers MUST allow each Intent pattern described in Appendix A 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 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 listed in Appendices A or B.

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. A list of required Broadcast Intents is provided in Appendix B; however, note that the SDK may define additional broadcast intents, which MUST also be honored.

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++**
- **OpenGL ES 1.1**

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 and binary compatibility.

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

3.4. Web API Compatibility

Many developers and applications rely on the behavior of the android.webkit.WebView class [Resources, 11] for their user interfaces, so the WebView implementation must be compatible across Android implementations. The Android Open Source implementation uses the WebKit rendering engine version to implement the WebView.

Because it is not feasible to develop a comprehensive test suite for a web browser, device implementers MUST use the specific upstream build of WebKit in the WebView implementation. Specifically:

- WebView MUST use the 528.5+ WebKit build from the upstream Android Open Source tree for Android 1.6. This build includes a specific set of functionality and security fixes for the WebView.
- The user agent string reported by the WebView MUST be in this format:
  `Mozilla/5.0 (Linux; U; Android 1.6; <language>-<country>; <device name>; Build/<build ID>) AppleWebKit/528.5+ (KHTML, like Gecko) Version/3.1.2 Mobile Safari/525.20.1`

- The "<device name>" string MUST be the same as the value for android.os.Build.MODEL
- The "<build ID>" string MUST be the same as the value for android.os.Build.ID.
- The "<language>" and "<country>" strings SHOULD follow the usual conventions for country code and language, and SHOULD refer to the curent locale of the device at the time of the request.

Implementations MAY ship a custom user agent string in the standalone Browser application. What's more, the standalone Browser MAY be based on an alternate browser technology (such as Firefox, Opera, etc.) However, even if an alternate Browser application is shipped, the WebView component provided to third-party applications MUST be based on WebKit, as above.

The standalone Browser application SHOULD include support for Gears [Resources, 12] and MAY include support for some or all of HTML5.

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 Android available from the Android Open Source Project.

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

A compatible Android device must support the full Dalvik Executable (DEX) bytecode specification and Dalvik Virtual Machine semantics [Resources, 13].

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, 14]. 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, 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, view, and remove AppWidgets.

3.8.2. Notifications

Android includes APIs that allow developers to notify users of notable events [Resources, 15]. 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 and all resources (icons, sound files, etc.) provided for in the APIs [Resources, 7], or in the Status Bar icon style guide [Resources, 16]. 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, 17] 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, 18]) 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.

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)
- ApiDemos (included in SDK)
- The "Apps for Android" applications [Resources, 19]

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

considered compatible.

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, 20].

Devices implementations MUST NOT extend either the .apk, Android Manifest, or Dalvik 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.

6. Multimedia Compatibility

A compatible Android device 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 [Resources, 4].

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.

Audio				
Name	**Encoder**	**Decoder**	**Details**	**Files Supported**
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) files. No support for raw AAC (.aac)
HE-AACv1 (AAC+)		X	Mono/Stereo content in any combination of standard bit rates up to 96 kbps and sampling rates between 8 to 48kHz	3GPP (.3gp) and MPEG-4 (.mp4, .m4a) files. No support for raw AAC (.aac)
HE-AACv2 (enhanced AAC+)		X	Mono/Stereo content in any combination of standard bit rates up to 96 kbps and sampling rates between 8 to 48kHz	3GPP (.3gp) and MPEG-4 (.mp4, .m4a) files. No support for raw AAC (.aac)
AMR-NB	X	X	4.75 to 12.2 kbps sampled @ 8kHz	3GPP (.3gp) files
AMR-WB		X	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16kHz	-3GPP (.3gp) files
MP3		X	Mono/Stereo 8-320Kbps constant (CBR) or variable bit-rate (VBR)	MP3 (.mp3) files
MIDI		X	MIDI Type 0 and 1. DLS Version 1 and 2. XMF and Mobile XMF.	Type 0 and 1 (.mid, .xmf, .mxmf). Also RTTTL/RTX (.rtttl, .rtx), OTA (.ota),

			Support for ringtone formats RTTTL/RTX, OTA, and iMelody	and iMelody (.imy)
Ogg Vorbis		X		.ogg
PCM		X	8- and 16-bit linear PCM (rates up to limit of hardware)	WAVE

Image				
Name	**Encoder**	**Decoder**	**Details**	**Files Supported**
JPEG	X	X	base+progressive	
GIF		X		
PNG	X	X		
BMP		X		

Video				
Name	**Encoder**	**Decoder**	**Details**	**Files Supported**
H.263	X	X		3GPP (.3gp) files
H.264		X		3GPP (.3gp) and MPEG-4 (.mp4) files
MPEG4 SP		X		3GPP (.3gp) file

7. Developer Tool Compatibility

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

- **Android Debug Bridge or adb** [Resources, 21]
  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 or ddms** [Resources, 22]
  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, 23]
  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 1.6 compatible devices must support. In Android 1.6, the majority of hardware features (such as WiFi, compass, and accelerometer) are required.

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.

8.1. Display

Android 1.6 includes facilities that perform certain automatic scaling and transformation operations under some circumstances, to ensure that third-party applications run reasonably well on hardware configurations for which they were not necessarily explicitly designed [Resources, 24]. Devices MUST properly implement these behaviors, as detailed in this section.

8.1.1. Standard Display Configurations

This table lists the standard screen configurations considered compatible with Android:

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, 25] 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 any resources that are present 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 8.2.1 require additional consideration and work to be compatible. Device implementers MUST contact Android Compatibility Team as provided for in Section 12 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 1.6; 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].

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

8.3. Non-touch Navigation

Device implementations:

- MAY omit non-touch navigation options (that is, may omit a trackball, 5-way directional pad, or wheel)
- MUST report via android.content.res.Configuration [Resources, 25] the correct value for the device's hardware

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 capacative 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

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 a USB mass storage client for the removable/media storage is present in the device
- SHOULD use the micro USB form factor on the device side
- 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)
- 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

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. WiFi

Device implementations MUST support 802.11b and 802.11g, and MAY support 802.11a.

8.9. Camera

Device implementations MUST include a camera. The included 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.

Device implementations MUST implement the following behaviors for the camera-related APIs [Resources, 27]:

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.

8.9.1. Non-Autofocus Cameras

If a device lacks an autofocus camera, the device implementer MUST meet the additional requirements in this section. Device implementations MUST implement the full Camera API included in the Android 1.6 SDK documentation in some reasonable way, regardless of actual camera hardware's capabilities.

For Android 1.6, if the camera lacks auto-focus, the device implementation MUST adhere to the following:

1. The system MUST include a read-only system property named "ro.workaround.noautofocus" with the value of "1". This value is intended to be used by applications such as Android Market to selectively identify device capabilities, and will be replaced in a future version of Android with a robust API.
2. If an application calls android.hardware.Camera.autoFocus(), the system MUST call the onAutoFocus() callback method on any registered android.hardware.Camera.AutoFocusCallback instances, even though no focusing actually happened. This is to avoid having existing applications break by waiting forever for an autofocus callback that will never come.
3. The call to the AutoFocusCallback.onAutoFocus() method MUST be triggered by the driver or framework in a new event on the main framework Looper thread. That is, Camera.autoFocus() MUST NOT directly call AutoFocusCallback.onAutoFocus() since this violates the Android framework threading model and will break apps.

8.10. Accelerometer

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

8.11. Compass

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

8.12. GPS

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

8.13. Telephony

Device implementations:

- MUST include either GSM or CDMA telephony
- MUST implement the appropriate APIs as detailed in the Android SDK documentation at developer.android.com

Note that this requirement implies that non-phone devices are not compatible with Android 1.6; Android 1.6 devices MUST include telephony hardware. Please see Appendix C for information on non-phone devices.

8.14. Volume controls

Android-compatible devices MUST include a mechanism to allow the user to increase and decrease the audio volume. Device implementations MUST make these functions available to the user at all times, regardless of application state. These functions MAY be implemented using physical hardware keys, software, gestures, touch panel, etc., but they MUST be always accessible and not obscure or interfere with the available application display area (see Display above).

When these buttons are used, the corresponding key events MUST be generated and sent to the foreground application. If the event is not intercepted and sunk by the application then device implementation MUST handle the event as a system volume control.

9. Performance Compatibility

One of the goals of the Android Compatibility Program is to ensure a consistent application experience for 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 1.6 compatible device, as in the table below:

Metric	Performance Threshold	Comments

Application Launch Time	The following applications should launch within the specified time. Browser: less than 1300ms MMS/SMS: less than 700ms AlarmClock: less than 650ms	This is tested by CTS. 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	Multiple applications will be launched. Re-launching the first application should complete taking less than the original launch time.	This is tested by CTS.

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 following security mechanisms:

10.1. Permissions

Device implementations MUST support the Android permissions model as defined in the Android developer documentation [Resources, 9]. 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. User 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].

11. Compatibility Test Suite

Device implementations MUST pass the Android Compatibility Test Suite (CTS) [Resources, 3] 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 1.6. However, such releases will only fix behavioral bugs in the CTS tests and will not impose any new tests, behaviors or APIs for a given platform release.

12. Contact Us

You can contact the Android Compatibility Team at compatibility@android.com for clarifications related to this Compatibiltiy Definition and to provide feedback on this Definition.

Appendix A: Required Application Intents

NOTE: this list is provisional, and will be updated in the future.

Application	Actions	Schemes	MIME Types
Browser	android.intent.action.VIEW	http https	(none) text/plain text/html application/xhtml+xml application/ vnd.wap.xhtml+xml
	android.intent.action.WEB_SEARCH	(none) http https	(none)
Camera	android.media.action.IMAGE_CAPTURE android.media.action.STILL_IMAGE_CAMERA android.media.action.VIDEO_CAMERA android.media.action.VIDEO_CAPTURE		
	android.intent.action.VIEW android.intent.action.GET_CONTENT android.intent.action.PICK android.intent.action.ATTACH_DATA		vnd.android.cursor.dir/ image vnd.android.cursor.dir/ video image/* video/*
	android.intent.action.VIEW	rtsp	
	android.intent.action.VIEW	http	video/mp4 video/3gp video/3gpp video/3gpp2
Phone / Contacts	android.intent.action.DIAL android.intent.action.VIEW android.intent.action.CALL	tel	
	android.intent.action.DIAL		
	android.intent.action.VIEW		vnd.android.cursor.dir/ person

	android.intent.action.PICK		vnd.android.cursor.dir/person vnd.android.cursor.dir/phone vnd.android.cursor.dir/postal-address
	android.intent.action.GET_CONTENT		vnd.android.cursor.item/person vnd.android.cursor.item/phone vnd.android.cursor.item/postal-address
Email	android.intent.action.SEND		text/plain image/* video/*
	android.intent.action.VIEW android.intent.action.SENDTO	mailto	
SMS / MMS	android.intent.action.VIEW android.intent.action.SENDTO	sms smsto mms mmsto	
Music	android.intent.action.VIEW	file	audio/* application/ogg application/x-ogg application/itunes
	android.intent.action.VIEW	http	audio/mp3 audio/x-mp3 audio/mpeg audio/mp4 audio/mp4a-latm
	android.intent.action.PICK		vnd.android.cursor.dir/artistalbum vnd.android.cursor.dir/album vnd.android.cursor.dir/nowplaying vnd.android.cursor.dir/track nd.android.cursor.dir/playlist vnd.android.cursor.dir/video
	android.intent.action.GET_CONTENT		media/* audio/* application/ogg application/x-ogg video/*

Package Installer	android.intent.action.VIEW	content file package	
	android.intent.action.PACKAGE_INSTALL	file http https	
	android.intent.action.ALL_APPS		
Settings	android.settings.SETTINGS android.settings.WIRELESS_SETTINGS android.settings.AIRPLANE_MODE_SETTINGS android.settings.WIFI_SETTINGS android.settings.APN_SETTINGS android.settings.BLUETOOTH_SETTINGS android.settings.DATE_SETTINGS android.settings.LOCALE_SETTINGS android.settings.INPUT_METHOD_SETTINGS com.android.settings.SOUND_SETTINGS com.android.settings.DISPLAY_SETTINGS android.settings.SECURITY_SETTING android.settings.LOCATION_SOURCE_SETTINGS android.settings.INTERNAL_STORAGE_SETTINGS android.settings.MEMORY_CARD_SETTINGS android.intent.action.SET_WALLPAPER		
Search	android.intent.action.SEARCH		query
	android.intent.action.SEARCH_LONG_PRESS		
Voice	android.intent.action.VOICE_COMMAND		

Contacts Management

Intent Action	Description
ATTACH_IMAGE	Starts an Activity that lets the user pick a contact to attach an image to.
EXTRA_CREATE_DESCRIPTION	Used with SHOW_OR_CREATE_CONTACT to specify an exact description to be

	shown when prompting user about creating a new contact.
EXTRA_FORCE_CREATE	Used with SHOW_OR_CREATE_CONTACT to force creating a new contact if no matching contact found.
SEARCH_SUGGESTION_CLICKED	This is the intent that is fired when a search suggestion is clicked on.
SEARCH_SUGGESTION_CREATE_CONTACT_CLICKED	This is the intent that is fired when a search suggestion for creating a contact is clicked on.
SEARCH_SUGGESTION_DIAL_NUMBER_CLICKED	This is the intent that is fired when a search suggestion for dialing a number is clicked on.
SHOW_OR_CREATE_CONTACT	Takes as input a data URI with a mailto: or tel: scheme.

Appendix B: Required Broadcast Intents **NOTE: this list is provisional, and will be updated in the future.**

Intent Action	Description
ACTION_BOOT_COMPLETED	Broadcast Action: This is broadcast once, after the system has finished booting.
ACTION_CALL_BUTTON	Broadcast Action: This is broadcast once, when a call is received.
ACTION_CAMERA_BUTTON	Broadcast Action: The "Camera Button" was pressed.
ACTION_CONFIGURATION_CHANGED	Broadcast Action: The current device Configuration (orientation, locale, etc) has changed.
ACTION_DATE_CHANGED	Broadcast Action: The date has changed.
ACTION_DEVICE_STORAGE_LOW	Broadcast Action: Indicates low memory condition on the device
ACTION_DEVICE_STORAGE_OK	Broadcast Action: Indicates low memory condition on the device no longer exists
ACTION_HEADSET_PLUG	Broadcast Action: Wired Headset plugged in or unplugged.
ACTION_INPUT_METHOD_CHANGED	Broadcast Action: An input method has been changed.
ACTION_MEDIA_BAD_REMOVAL	Broadcast Action: External media was removed from SD card slot, but mount point was not unmounted.
ACTION_MEDIA_BUTTON	Broadcast Action: The "Media Button" was pressed.
ACTION_MEDIA_CHECKING	Broadcast Action: External media is present, and being disk-checked The path to the mount point for the checking media is contained in the Intent.mData field.
ACTION_MEDIA_EJECT	Broadcast Action: User has expressed the desire to remove the external storage media.
ACTION_MEDIA_MOUNTED	Broadcast Action: External media is present and mounted at its mount point.
ACTION_MEDIA_NOFS	Broadcast Action: External media is present, but is using an incompatible fs (or is blank) The path to the mount point for the checking media is contained in the Intent.mData field.
ACTION_MEDIA_REMOVED	Broadcast Action: External media has been removed.
ACTION_MEDIA_SCANNER_FINISHED	Broadcast Action: The media scanner has finished scanning a directory.
ACTION_MEDIA_SCANNER_SCAN_FILE	Broadcast Action: Request the media scanner to scan a file and add it to the media database.

ACTION_MEDIA_SCANNER_STARTED	Broadcast Action: The media scanner has started scanning a directory.
ACTION_MEDIA_SHARED	Broadcast Action: External media is unmounted because it is being shared via USB mass storage.
ACTION_MEDIA_UNMOUNTABLE	Broadcast Action: External media is present but cannot be mounted.
ACTION_MEDIA_UNMOUNTED	Broadcast Action: External media is present, but not mounted at its mount point.
ACTION_NEW_OUTGOING_CALL	Broadcast Action: An outgoing call is about to be placed.
ACTION_PACKAGE_ADDED	Broadcast Action: A new application package has been installed on the device.
ACTION_PACKAGE_CHANGED	Broadcast Action: An existing application package has been changed (e.g. a component has been enabled or disabled.
ACTION_PACKAGE_DATA_CLEARED	Broadcast Action: The user has cleared the data of a package. This should be preceded by ACTION_PACKAGE_RESTARTED, after which all of its persistent data is erased and this broadcast sent. Note that the cleared package does *not* receive this broadcast. The data contains the name of the package.
ACTION_PACKAGE_REMOVED	Broadcast Action: An existing application package has been removed from the device. The data contains the name of the package. The package that is being installed does *not* receive this Intent.
ACTION_PACKAGE_REPLACED	Broadcast Action: A new version of an application package has been installed, replacing an existing version that was previously installed.
ACTION_PACKAGE_RESTARTED	Broadcast Action: The user has restarted a package, and all of its processes have been killed. All runtime state associated with it (processes, alarms, notifications, etc) should be removed. Note that the restarted package does *not* receive this broadcast. The data contains the name of the package.
ACTION_PROVIDER_CHANGED	Broadcast Action: Some content providers have parts of their namespace where they publish new events or items that the user may be especially interested in.
ACTION_SCREEN_OFF	Broadcast Action: Sent after the screen turns off.
ACTION_SCREEN_ON	Broadcast Action: Sent after the screen turns on.
ACTION_UID_REMOVED	Broadcast Action: A user ID has been removed from the system.
ACTION_UMS_CONNECTED	Broadcast Action: The device has entered USB Mass Storage mode.

ACTION_UMS_DISCONNECTED	Broadcast Action: The device has exited USB Mass Storage mode.
ACTION_USER_PRESENT	Broadcast Action: Sent when the user is present after device wakes up (e.g when the keyguard is gone).
ACTION_WALLPAPER_CHANGED	Broadcast Action: The current system wallpaper has changed.
ACTION_TIME_CHANGED	Broadcast Action: The time was set.
ACTION_TIME_TICK	Broadcast Action: The current time has changed.
ACTION_TIMEZONE_CHANGED	Broadcast Action: The timezone has changed.
ACTION_BATTERY_CHANGED	Broadcast Action: The charging state, or charge level of the battery has changed.
ACTION_BATTERY_LOW	Broadcast Action: Indicates low battery condition on the device. This broadcast corresponds to the "Low battery warning" system dialog.
ACTION_BATTERY_OKAY	Broadcast Action: Indicates the battery is now okay after being low. This will be sent after ACTION_BATTERY_LOW once the battery has gone back up to an okay state.

Network State

Intent Action	Description
NETWORK_STATE_CHANGED_ACTION	Broadcast intent action indicating that the state of Wi-Fi connectivity has changed.
RSSI_CHANGED_ACTION	Broadcast intent action indicating that the RSSI (signal strength) has changed.
SUPPLICANT_STATE_CHANGED_ACTION	Broadcast intent action indicating that a connection to the supplicant has been established or lost.
WIFI_STATE_CHANGED_ACTION	Broadcast intent action indicating that Wi-Fi has been enabled, disabled, enabling, disabling, or unknown.
NETWORK_IDS_CHANGED_ACTION	The network IDs of the configured networks could have changed.
ACTION_BACKGROUND_DATA_SETTING_CHANGED	Broadcast intent action indicating that the setting for background data usage has changed values.
CONNECTIVITY_ACTION	Broadcast intent indicating that a change in network connectivity has occurred.
ACTION_AIRPLANE_MODE_CHANGED	Broadcast Action: The user has switched the phone into or out of Airplane Mode.

Appendix C: Future Considerations

This appendix clarifies certain portions of this Android 1.6 Compatibility Definition, and in some cases discusses anticipated or planned changes intended for a future version of the Android platform. This appendix is for informational and planning purposes only, and is not part of the Compatibility Definition for Android 1.6.

1. Non-telephone Devices

Android 1.6 is intended exclusively for telephones; telephony functionality is not optional. Future versions of the Android platform are expected to make telephony optional (and thus allow for non-phone Android devices), but only phones are compatible with Android 1.6.

2. Bluetooth Compatibility

The Android 1.6 release of Android does not support Bluetooth APIs, so from a compatibility perspective Bluetooth does not impose any considerations for this version of the platform. However, a future version of Android will introduce Bluetooth APIs. At that point, supporting Bluetooth will become mandatory for compatibility.

Consequently, we strongly recommend that Android 1.6 devices include Bluetooth, so that they will be compatible with future versions of Android that require Bluetooth.

3. Required Hardware Components

All hardware components in Section 8 (including WiFi, magnetometer/compass, accelerometer, etc.) are required and may not be omitted. Future versions of Android are expected to make some (but not all) of these components optional, in tandem with corresponding tools for third-party developers to handle these changes.

4. Sample Applications

The Compatibility Definition Document for a future version of Android will include a more extensive and representative list of applications than the ones listed in Section 4, above. For Android 1.6, the applications listed in Section 4 must be tested.

5. Touch Screens

Future versions of the Compatibility Definition may or may not allow for devices to omit touchscreens. However, currently much of the Android framework implementation assumes the existence of a touchscreen; omitting a touchscreen would break substantially all current third-party Android applications, so in Android 1.6 a touchscreen is required for compatibility.

6. Performance

Future versions of CTS will also measure the CPU utilization and performance of the following components of an implementation:

- 2D graphics
- 3D graphics
- Video playback
- Audio playback
- Bluetooth A2DP playback

Android 2.1 Compatibility Definition

compatibility@android.com

1. Introduction

This document enumerates the requirements that must be met in order for mobile phones to be compatible with Android 2.1.

The use of "must", "must not", "required", "shall", "shall not", "should", "should not", "recommended", "may" and "optional" is per the IETF standard defined in RFC2119 [Resources, 1].

As used in this document, a "device implementer" or "implementer" is a person or organization developing a hardware/software solution running Android 2.1. A "device implementation" or "implementation" is the hardware/software solution so developed.

To be considered compatible with Android 2.1, 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 2.1 allowed version strings: http://source.android.com/compatibility/2.1/versions.xhtml
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. Supporting Multiple Screens: http://developer.android.com/guide/practices/screens_support.html
24. android.content.res.Configuration: http://developer.android.com/reference/android/content/res/Configuration.html
25. android.util.DisplayMetrics: http://developer.android.com/reference/android

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

Many of these resources are derived directly or indirectly from the Android 2.1 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 2.1 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 2.1. 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 addtional 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 2.1, this field MUST have the integer value 7.
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 shipped 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:2.1-update1/ERC77`
	`/3359:userdebug/test-keys`
	The fingerprint MUST NOT include spaces. If other fields included in the template above have spaces, they SHOULD be replaced with the ASCII underscore ("_") character in the fingerprint.

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 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 implementers MUST allow each Intent pattern defined in core system apps 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.

Note: this section was modified by Erratum EX6580.

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 API 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. The Android Open Source implementation uses the WebKit rendering engine to implement the WebView.

Because it is not feasible to develop a comprehensive test suite for a web browser, device implementers MUST use the specific upstream build of WebKit in the WebView implementation. Specifically:

- WebView MUST use the 530.17 WebKit build from the upstream Android Open Source tree for Android 2.1. This build includes a specific set of functionality and security fixes for the WebView.
- The user agent string reported by the WebView MUST be in this format:
  `Mozilla/5.0 (Linux; U; Android $(VERSION); $(LOCALE); $(MODEL) Build/$(BUILD)) AppleWebKit/530.17 (KHTML, like Gecko) Version/4.0 Mobile Safari/530.17`
  - 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`

Implementations MAY ship a custom user agent string in the standalone Browser application. What's more, the standalone Browser MAY be based on an alternate browser technology (such as Firefox, Opera, etc.) However, even if an alternate Browser application is shipped, the WebView component provided to third-party applications MUST be based on WebKit, as above.

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 in some form. The standalone Browser application (whether based on the upstream WebKit Browser application or a third-party replacement) MUST include support for the same HTML5 features just listed for WebView.

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 MUST configure Dalvik to allocate at least 16MB of memory to each application on devices with screens classified as medium- or low-density. Device implementations MUST configure Dalvik to allocate at least 24MB of memory to each application on devices with screens classified as high-density. Note that device implementations MAY allocate more memory than these figures, but are not required to.

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 implemenations 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].

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 implemenations 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)

	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.

7. Developer Tool Compatibility

Device implemenations 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 2.1 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 accurate report accurate hardware configuration information via the `getSystemAvailableFeatures()` and `hasSystemFeature(String)` methods on the `android.content.pm.PackageManager` class.

8.1. Display

Android 2.1 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, 23]. Devices MUST properly implement these behaviors, as detailed in this section.

For Android 2.1, this 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 provided for in Section 12 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 2.1; 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 valuesfor all display metrics defined in `android.util.DisplayMetrics` [Resources, 25].

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, 24] (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, 24]

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 capacative or resistive touchscreen
- MUST report the value of `android.content.res.Configuration` [Resources, 24] reflecting corresponding to the type of the specific touchscreen on the device

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

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 camera. The included 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 2.1 SDK documentation [Resources, 26]), 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, unless the constants are prefixed with a string indicating the name of the device implementer. That is, device implementations MUST support all standard Camera parameters if the hardware allows, and MUST NOT support custom Camera parameter types unless the parameter names are clearly indicated via a string prefix to be non-standard.

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, 27]).

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, 27]).

8.12. GPS

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

8.13. Telephony

Android 2.1 MAY be used on devices that do not include telephony hardware. That is, Android 2.1 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.

Note: this section was modified by Erratum EX6580.

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 and mounted on `/sdcard` (or `/sdcard` MUST be a symbolic link to the physical location if it is mounted elsewhere.)

Note: this section was added by Erratum EX6580.

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, 29]. Device implementations SHOULD implement relevant Bluetooth profiles, such as A2DP, AVRCP, OBEX, etc. as appropriate for the device.

Note: this section was added by Erratum EX6580.

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 2.1 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	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

	• MMS/SMS: less than 700ms • AlarmClock: less than 650ms	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, 28] 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, 28]. 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, 28].

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, 28].

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 2.1. 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 thid-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: Android 2.1 Compatibility Definition, Erratum EX6580

The Android 2.1 Compatibility Definition is modified by this erratum, as follows.

Section 3.2.3.2. Intent Overrides

Section 3.2.3.2 on Intent Overrides is modified to remove the references to the non-existent "Appendix A". The revised text follows.

As Android is an extensible platform, device implementers MUST allow each Intent pattern defined in core system apps 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.

Section 8.14. Memory and Storage

Section 3.2.3.2 on Memory and Storage is modified to correct erroneous values provided for minimum internal storage requirements. The minimum internal storage is revised downward, from 290MB to 150MB. The revised text follows.

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.

Section 8.15. Application Shared Storage

Section 8.15 is added, to clarify existing requirements on external/shared storage. The added text follows.

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 and mounted on `/sdcard` (or `/sdcard` MUST be a symbolic link to the physical location if it is mounted elsewhere.)

Section 8.16. Bluetooth

Section 8.16 is added, to clarify existing requirements on the inclusion of Bluetooth hardware. The added text follows.

Device implementations MUST include a Bluetooth transceiver. Device implementations MUST enable the RFCOMM-based Bluetooth API as described in the SDK documentation [Resources, 29]. Device implementations SHOULD implement relevant Bluetooth profiles, such as A2DP, AVRCP, OBEX, etc. as

appropriate for the device.

android compatibility program

Android 2.2 Compatibility Definition

compatibility@android.com

Table of Contents

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
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 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/myproduct/mydevice:generic:2.2/FRG77/3359: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 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 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 valuesfor 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 capacative 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	The following applications should launch within the specified time. • 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 thid-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.

android compatibility program

Android 2.3 Compatibility Definition

compatibility@android.com

Table of Contents

1. Introduction

This document enumerates the requirements that must be met in order for mobile phones to be compatible with Android 2.3.

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

To be considered compatible with Android 2.3, 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.

Please note that this Compatibility Definition is issued to correspond with the 2.3.3 update to Android, which is API level 10. This Definition obsoletes and replaces the Compatibility Definition for Android 2.3 versions prior to 2.3.3. (That is, versions 2.3.1 and 2.3.2 are obsolete.) Future Android-compatible devices running Android 2.3 MUST ship with version 2.3.3 or later.

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 2.3 allowed version strings: http://source.android.com/compatibility/2.3/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. HTML5 offline capabilities: http://dev.w3.org/html5/spec/Overview.html#offline
11. HTML5 video tag: http://dev.w3.org/html5/spec/Overview.html#video
12. HTML5/W3C geolocation API: http://www.w3.org/TR/geolocation-API/
13. HTML5/W3C webdatabase API: http://www.w3.org/TR/webdatabase/
14. HTML5/W3C IndexedDB API: http://www.w3.org/TR/IndexedDB/
15. Dalvik Virtual Machine specification: available in the Android source code, at dalvik/docs
16. AppWidgets: http://developer.android.com/guide/practices/ui_guidelines/widget_design.html
17. Notifications: http://developer.android.com/guide/topics/ui/notifiers/notifications.html
18. Application Resources: http://code.google.com/android/reference/available-resources.html
19. Status Bar icon style guide: http://developer.android.com/guide/practices/ui_guideline /icon_design.html#statusbarstructure
20. Search Manager: http://developer.android.com/reference/android/app/SearchManager.html
21. Toasts: http://developer.android.com/reference/android/widget/Toast.html
22. Live Wallpapers: http://developer.android.com/resources/articles/live-wallpapers.html
23. Reference tool documentation (for adb, aapt, ddms): http://developer.android.com/guide/developing/tools/index.html
24. Android apk file description: http://developer.android.com/guide/topics/fundamentals.html
25. Manifest files: http://developer.android.com/guide/topics/manifest/manifest-intro.html

26. Monkey testing tool: http://developer.android.com/guide/developing/tools/monkey.html
27. Android Hardware Features List: http://developer.android.com/reference/android/content/pm/PackageManager.html
28. Supporting Multiple Screens: http://developer.android.com/guide/practices/screens_support.html
29. android.util.DisplayMetrics: http://developer.android.com/reference/android/util/DisplayMetrics.html
30. android.content.res.Configuration: http://developer.android.com/reference/android/content/res/Configuration.html
31. Sensor coordinate space: http://developer.android.com/reference/android/hardware/SensorEvent.html
32. Bluetooth API: http://developer.android.com/reference/android/bluetooth/package-summary.html
33. NDEF Push Protocol: http://source.android.com/compatibility/ndef-push-protocol.pdf
34. MIFARE MF1S503X: http://www.nxp.com/documents/data_sheet/MF1S503x.pdf
35. MIFARE MF1S703X: http://www.nxp.com/documents/data_sheet/MF1S703x.pdf
36. MIFARE MF0ICU1: http://www.nxp.com/documents/data_sheet/MF0ICU1.pdf
37. MIFARE MF0ICU2: http://www.nxp.com/documents/short_data_sheet/MF0ICU2_SDS.pdf
38. MIFARE AN130511: http://www.nxp.com/documents/application_note/AN130511.pdf
39. MIFARE AN130411: http://www.nxp.com/documents/application_note/AN130411.pdf
40. Camera orientation API: http://developer.android.com/reference/android/hardware/Camera.html#setDisplayOrientation(int)
41. android.hardware.Camera: http://developer.android.com/reference/android/hardware/Camera.html
42. Android Security and Permissions reference: http://developer.android.com/guide/topics/security/security.html
43. Apps for Android: http://code.google.com/p/apps-for-android

Many of these resources are derived directly or indirectly from the Android 2.3 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 2.3 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. This section details the "soft" APIs and system behaviors required for compatibility with Android 2.3. 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
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 2.3, this field MUST have the integer value 9.
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.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/generic:2.3/GRC77/3359: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.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.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. The value of this field MUST be encodable as 7-bit ASCII and match the regular expression "^[a-zA-Z0-9.,_-]+$".
android.os.Build.TAGS	A comma-separated list of tags chosen by the device implementer that further distinguish the build. For example, "unsigned,debug". The value of this field MUST be encodable as 7-bit ASCII and match the regular expression "^[a-zA-Z0-9.,_-]+$".
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

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
- Contacts
- Email
- 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 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. 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.txt`. 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 (Open Sound Library audio 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

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 complete, modern web broswer 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 2.3. 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 component SHOULD include support for as much of HTML5 [Resources, 9] as possible. Minimally, device implementations MUST support each of these APIs associated with HTML5 in the WebView:

- application cache/offline operation [Resources, 10]
- the <video> tag [Resources, 11]
- geolocation [Resources, 12]

Additionally, device implementations MUST support the HTML5/W3C webstorage API [Resources, 13], and SHOULD support the HTML5/W3C IndexedDB API [Resources, 14]. *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, 9] as possible. Minimally, device implementations MUST support each of these APIs associated with HTML5:

- application cache/offline operation [Resources, 10]
- the <video> tag [Resources, 11]
- geolocation [Resources, 12]

Additionally, device implementations MUST support the HTML5/W3C webstorage API [Resources, 13], and SHOULD support the HTML5/W3C IndexedDB API [Resources, 14]. *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, 15].

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 or extra-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, 16]. 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, 17]. 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, 18], or in the Status Bar icon style guide [Resources, 19]. 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, 20] 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, 21]) 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, 22]. 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. 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, 23].

Devices implementations MUST NOT extend either the .apk [Resources, 24], Android Manifest [Resources, 25], or Dalvik bytecode [Resources, 15] 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 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. 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 multimedia codecs as detailed in the following sections. 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.

The tables below do 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.

5.1.1. Media Decoders

Device implementations MUST include an implementation of an decoder for each codec and format described in the table below. Note that decoders for each of these media types are provided by the upstream Android Open-Source Project.

	Name	Details	File/Container Format
Audio	AAC LC/LTP	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+)		
	HE-AACv2 (enhanced AAC+)		
	AMR-NB	4.75 to 12.2 kbps sampled @ 8kHz	3GPP (.3gp)
	AMR-WB	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16kHz	3GPP (.3gp)
	MP3	Mono/Stereo 8-320Kbps constant (CBR) or variable bit-rate (VBR)	MP3 (.mp3)
	MIDI	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		Ogg (.ogg)
	PCM	8- and 16-bit linear PCM (rates up to limit of hardware)	WAVE (.wav)
Image	JPEG	base+progressive	
	GIF		
	PNG		
	BMP		
Video	H.263		3GPP (.3gp) files
	H.264		3GPP (.3gp) and MPEG-4 (.mp4) files
	MPEG4 Simple Profile		3GPP (.3gp) file

5.1.2. Media Encoders

Device implementations SHOULD include encoders for as many of the media formats listed in Section 5.1.1. as possible. However, some encoders do not make sense for devices that lack certain optional hardware; for instance, an encoder for the H.263 video does not make sense, if the device lacks any cameras. Device implementations MUST therefore implement media encoders according to the conditions described in the table below.

See Section 7 for details on the conditions under which hardware may be omitted by device implementations.

	Name	Details	File/Container Format	Conditions
Audio	AMR-NB	4.75 to 12.2 kbps sampled @ 8kHz	3GPP (.3gp)	Device implementations that include microphone hardware and define `android.hardware.microphone` MUST include encoders for these audio formats.
	AMR-WB	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16kHz	3GPP (.3gp)	
	AAC LC/LTP	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).	

Image	JPEG	base+progressive		All device implementations MUST include encoders for these image formats, as Android 2.3 includes APIs that applications can use to programmatically generate files of these types.
	PNG			
Video	H.263		3GPP (.3gp) files	Device implementations that include camera hardware and define either `android.hardware.camera` or `android.hardware.camera.front` MUST include encoders for these video formats.

In addition to the encoders listed above, device implementations SHOULD include an H.264 encoder. Note that the Compatibility Definition for a future version is planned to change this requirement to "MUST". That is, H.264 encoding is optional in Android 2.3 but **will be required** by a future version. Existing and new devices that run Android 2.3 are **very strongly encouraged to meet this requirement in Android 2.3**, or they will not be able to attain Android compatibility when upgraded to the future version.

5.2. 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 2.3, the Compatibility Definition for a future version is planned to change these to "MUST". That is, these requirements are optional in Android 2.3 but **will be required** by a future version. Existing and new devices that run Android 2.3 are **very strongly encouraged to meet these requirements in Android 2.3**, or they will not be able to attain Android compatibility when upgraded to the future version.

5.3. 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 that include microphone hardware and declare `android.hardware.microphone` SHOULD meet all audio latency requirements outlined in this section. See Section 7 for details on the conditions under which microphone hardware may be omitted by device implementations.

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 2.3, the Compatibility Definition for a future version is planned to change these to "MUST". That is, these requirements are optional in Android 2.3 but **will be required** by a future version. Existing and new devices that run Android 2.3 are **very strongly encouraged to meet these requirements in Android 2.3**, or they will not be able to attain Android compatibility when upgraded to the future version.

If a device implementation meets the requirements of this section, 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, 27] Conversely, if the device implementation does not meet these requirements it MUST NOT report support for low-latency audio.

6. 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, 23]
  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, 23]
  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, 26]
  Device implementations MUST include the Monkey framework, and make it available for applications to use.

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, and Windows 7, in both 32-bit and 64-bit versions.

7. Hardware Compatibility

Android is intended to enable device implementers to create innovative form factors and configurations. At the same time Android developers write innovative applications that rely on the various hardware and features available through the Android APIs. The requirements in this section strike a balance between innovations available to device implementers, and the needs of developers to ensure their apps are only available to devices where they will run properly.

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, 27]

7.1. Display and Graphics

Android 2.3 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, 28]. Devices MUST properly implement these APIs and behaviors, as detailed in this section.

7.1.1. Screen Configurations

Device implementations MAY use screens of any pixel dimensions, provided that they meet the following requirements:

- screens MUST be at least 2.5 inches in physical diagonal size
- density MUST be at least 100 dpi
- the aspect ratio MUST be between 1.333 (4:3) and 1.779 (16:9)
- the display technology used consists of square pixels

Device implementations with a screen meeting the requirements above are considered compatible, and no additional action is necessary. The Android framework implementation automatically computes display characteristics such as screen size bucket and density bucket. In the majority of cases, the framework decisions are the correct ones. If the default framework computations are used, no additional action is necessary. Device implementers wishing to change the defaults, or use a screen that does not meet the requirements above MUST contact the Android Compatibility Team for guidance, as provided for in Section 12.

The units used by the requirements above 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".

Device implementations MUST use only displays with a single static configuration. That is, device implementations MUST NOT enable multiple screen configurations. For instance, since a typical television supports multiple resolutions such as 1080p, 720p, and so on, this configuration is not compatible with Android 2.3. (However, support for such configurations is under investigation and planned for a future version of Android.)

7.1.2. Display Metrics

Device implementations MUST report correct values for all display metrics defined in `android.util.DisplayMetrics` [Resources, 29].

7.1.3. Declared Screen Support

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

7.1.4. 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 that cannot be physically rotated MAY meet this requirement by "letterboxing" applications that request portrait mode, using only a portion of the available display.

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.

7.1.5. 3D Graphics Acceleration

Device implementations MUST support OpenGL ES 1.0, as required by the Android 2.3 APIs. For devices that lack 3D acceleration hardware, a software implementation of OpenGL ES 1.0 is provided by the upstream Android Open-Source Project. Device implementations SHOULD support OpenGL ES 2.0.

Implementations MAY omit Open GL ES 2.0 support; however if support is ommitted, device implementations MUST NOT report as supporting OpenGL ES 2.0. Specifically, if a device implementations lacks OpenGL ES 2.0 support:

- the managed APIs (such as via the `GLES10.getString()` method) MUST NOT report support for OpenGL ES 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 NOT report support for OpenGL ES 2.0.

Conversely, if a device implementation *does* support OpenGL ES 2.0, it MUST accurately report that support via the routes just listed.

Note that Android 2.3 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 2.3 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.

7.2. Input Devices

Android 2.3 supports a number of modalities for user input. Device implementations MUST support user input devices as provided for in this section.

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 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, 30] (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, 30]
- 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 code 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, 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.

7.2.4. 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, 30] reflecting corresponding to the type of the specific touchscreen on the device
- SHOULD support fully independently tracked pointers, if the touchscreen supports multiple pointers

7.3. Sensors

Android 2.3 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, 27]
- 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.)

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 2.3 APIs introduce 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 2.3 SDK documentation to be a streaming sensor.

7.3.1. Accelerometer

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

- MUST be able to deliver events at 50 Hz or greater
- MUST comply with the Android sensor coordinate system as detailed in the Android APIs (see [Resources, 31])
- 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, 31]).
- 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 capable of measuring orientation changes up to 5.5*Pi radians/second (that is, approximately 1,000 degrees per second)

- MUST be able to deliver events at 100 Hz or greater
- MUST have 8-bits of accuracy or more

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

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 2.3 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

Network connectivity and access to the Internet are vital features of Android. Meanwhile, device-to-device interaction adds significant value to Android devices and applications. Device implementations MUST meet the data connectivity requirements in this section.

7.4.1. Telephony

"Telephony" as used by the Android 2.3 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 2.3 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 2.3 MAY be used on devices that do not include telephony hardware. That is, Android 2.3 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 2.3 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.

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, 32]. 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, 27]
- 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)
    - NfcV (ISO 15693)
    - IsoDep (ISO 14443-4)
    - NFC Forum Tag Types 1, 2, 3, 4 (defined by the NFC Forum)
  - 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, 33]
  - MUST scan for all supported technologies while in NFC discovery mode.
  - SHOULD be in NFC discovery mode while the device is awake with the screen active.

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

Additionally, device implementations SHOULD support the following widely-deployed MIFARE technologies.

- MIFARE Classic (NXP MF1S503x [Resources, 34], MF1S703x [Resources, 35])
- MIFARE Ultralight (NXP MF0ICU1 [Resources, 36], MF0ICU2 [Resources, 37])
- NDEF on MIFARE Classic (NXP AN130511 [Resources, 38], AN130411 [Resources, 39])

Note that Android 2.3.3 includes APIs for these MIFARE types. If a device implementation supports MIFARE, 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, 27] 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, 27], and MUST implement the Android 2.3 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 2.3 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, 40] 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 data returned to any "postview" camera callback handlers, in the same manner as the camera preview image stream. (If the device implementation does not support postview callbacks, 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 SHOULD support the YV12 format (as denoted by the `android.graphics.ImageFormat.YV12` constant) for camera previews for both front- and rear-facing cameras. Note that the Compatibility Definition for a future version is planned to change this requirement to "MUST". That is, YV12 support is optional in Android 2.3 but **will be required** by a future version. Existing and new devices that run Android 2.3 are **very strongly encouraged to meet this requirement in Android 2.3**, or they will not be able to attain Android compatibility when upgraded to the future version.

Device implementations MUST implement the full Camera API included in the Android 2.3 SDK documentation [Resources, 41]), 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.

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, a 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

The fundamental function of Android 2.3 is to run applications. Device implementations MUST the requirements of this section, to ensure adequate storage and memory for applications to run properly.

7.6.1. Minimum Memory and Storage

Device implementations MUST have at least 128MB of memory available to the kernel and userspace. The 128MB 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 1GB of non-volatile storage available for user data. Note that this higher requirement is planned to become a hard minimum 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.

The Android APIs include a Download Manager that applications may use to download data files. The Download Manager implementation MUST be capable of downloading individual files 55MB in size, or larger. The Download Manager implementation SHOULD be capable of downloading files 100MB in size, or larger.

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 or Media Transfer Protocol.

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:

- 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

8. Performance Compatibility

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 2.3 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 • 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.	

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, 42] 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, 42]. 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, 42].

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, 42].

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.

10. Software Compatibility Testing

The Android Open-Source Project includes various testing tools to verify that device implementations are compatible. 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 2.3 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 2.3. Device implementations MUST pass the latest CTS version available at the time the device software is completed.

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.

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, 43].
- 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.

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. 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

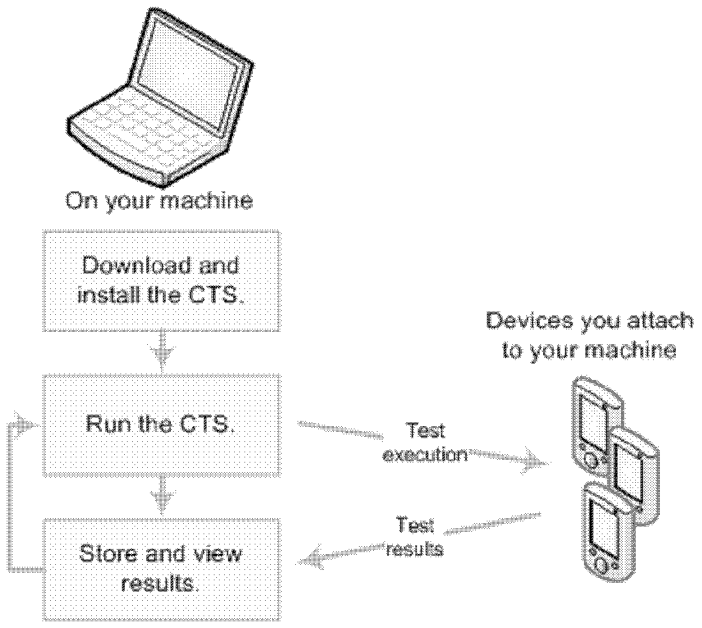
1. Re-launch the Bluetooth Chat app on both devices.
2. Send 10 or messages from each device, and verify that the other device receives them correctly.

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.

Compatibility Test Suite

How does the CTS work?



The CTS is an automated testing harness that includes two major software components:

- The CTS test harness runs on your desktop machine and manages test execution.
- Individual test cases are executed on attached mobile devices or on an emulator. The test cases are written in Java as JUnit tests and packaged as Android .apk files to run on the actual device target.

Workflow

1. Download the CTS.
2. Attach at least one device (or emulator) to your machine.
3. For CTS 2.1 R2 and beyond, setup your device (or emulator) to run the accessibility tests:
    1. adb install -r android-cts/repository/testcases/CtsDelegatingAccessibilityService.apk
    2. On the device, enable Settings > Accessibility > Accessibility > Delegating Accessibility Service
4. For CTS 2.3 R4 and beyond, setup your device to run the device administration tests:
    1. adb install -r android-cts/repository/testcases/CtsDeviceAdmin.apk
    2. On the device, enable all the android.deviceadmin.cts.* device administrators under Settings > Location & security > Select device administrators
5. Launch the CTS. The CTS test harness loads the test plan onto the attached devices. For each test in the test harness:
    - The test harness pushes a .apk file to each device, executes the test through instrumentation, and

records test results.

- The test harness removes the .apk file from each device.

6. Once all the tests are executed, you can view the test results in your browser and use the results to adjust your design. You can continue to run the CTS throughout your development process.

When you are ready, you can submit the report generated by the CTS to cts@android.com. The report is a .zip archived file that contains XML results and supplemental information such as screen captures.

Types of test cases

The CTS includes the following types of test cases:

- *Unit tests* test atomic units of code within the Android platform; e.g. a single class, such as java.util.HashMap.
- *Functional tests* test a combination of APIs together in a higher-level use-case.
- *Reference application tests* instrument a complete sample application to exercise a full set of APIs and Android runtime services

Future versions of the CTS will include the following types of test cases:

- *Robustness tests* test the durability of the system under stress.
- *Performance tests* test the performance of the system against defined benchmarks, for example rendering frames per second.

Areas Covered

The unit test cases cover the following areas to ensure compatibility:

Area	Description
Signature tests	For each Android release, there are XML files describing the signatures of all public APIs contained in the release. The CTS contains a utility to check those API signatures against the APIs available on the device. The results from signature checking are recorded in the test result XML file.
Platform API Tests	Test the platform (core libraries and Android Application Framework) APIs as documented in the SDK Class Index to ensure API correctness, including correct class, attribute and method signatures, correct method behavior, and negative tests to ensure expected behavior for incorrect parameter handling.
Dalvik VM Tests	The tests focus on testing the Dalvik VM
Platform Data Model	The CTS tests the core platform data model as exposed to application developers through content providers, as documented in the SDK android.provider package: contacts, browser, settings, etc.
Platform Intents	The CTS tests the core platform intents, as documented in the SDK Available Intents.
Platform Permissions	The CTS tests the core platform permissions, as documented in the SDK Available Permissions.

Platform Resources	The CTS tests for correct handling of the core platform resource types, as documented in the SDK Available Resource Types. This includes tests for: simple values, drawables, nine-patch, animations, layouts, styles and themes, and loading alternate resources.

CTS Development

Initializing Your Repo Client

Follow the instructions to get and build the Android source code but specify `-b gingerbread` when issuing the `repo init` command. This assures that your CTS changes will be included in the next CTS release and beyond.

Setting Up Eclipse

Follow the instructions to setup Eclipse but execute the following command to generate the `.classpath` file rather than copying the one from the development project:

```
cd /path/to/android/root
./cts/development/ide/eclipse/genclasspath.sh > .classpath
chmod u+w .classpath
```

This `.classpath` file will contain both the Android framework packages and the CTS packages.

Building and Running CTS

Execute the following commands to build CTS and start the interactive CTS console:

```
cd /path/to/android/root
make cts
cts
```

Provide arguments to CTS to immediately start executing a test:

```
cts start --plan CTS -p android.os.cts.BuildVersionTest
```

Writing CTS Tests

CTS tests use JUnit and the Android testing APIs. Review the Testing and Instrumentation tutorial while perusing the existing tests under the `cts/tests/tests` directory. You will see that CTS tests mostly follow the same conventions used in other Android tests.

Since CTS runs across many production devices, the tests must follow these rules:

- Must take into account varying screen sizes, orientations, and keyboard layouts.
- Only use public API methods. In other words, avoid all classes, methods, and fields that are annotated with the "hide" annotation.
- Avoid relying upon particular view layouts or depend on the dimensions of assets that may not be on some device.

- Don't rely upon root privileges.

Test Naming and Location

Most CTS test cases target a specific class in the Android API. These tests have Java package names with a `cts` suffix and class names with the `Test` suffix. Each test case consists of multiple tests, where each test usually exercises a particular API method of the API class being tested. These tests are arranged in a directory structure where tests are grouped into different categories like "widgets" and "views."

For example, the CTS test for `android.widget.TextView` is `android.widget.cts.TextViewTest` found under the `cts/tests/tests/widget/src/android/widget/cts` directory with its Java package name as `android.widget.cts` and its class name as `TextViewTest`. The `TextViewTest` class has a test called `testSetText` that exercises the "setText" method and a test named "testSetSingleLine" that calls the `setSingleLine` method. Each of those tests have `@TestTargetNew` annotations indicating what they cover.

Some CTS tests do not directly correspond to an API class but are placed in the most related package possible. For instance, the CTS test, `android.net.cts.ListeningPortsTest`, is in the `android.net.cts`, because it is network related even though there is no `android.net.ListeningPorts` class. You can also create a new test package if necessary. For example, there is an "android.security" test package for tests related to security. Thus, use your best judgement when adding new tests and refer to other tests as examples.

Finally, a lot of tests are annotated with @TestTargets and @TestTargetNew. These are no longer necessary so do not annotate new tests with these.

New Test Packages

When adding new tests, there may not be an existing directory to place your test. In that case, refer to the example under `cts/tests/tests/example` and create a new directory. Furthermore, make sure to add your new package's module name from its `Android.mk` to `CTS_COVERAGE_TEST_CASE_LIST` in `cts/CtsTestCaseList.mk`. This Makefile is used by `build/core/tasks/cts.mk` to glue all the tests together to create the final CTS package.

Test Stubs and Utilities

Some tests use additional infrastructure like separate activities and various utilities to perform tests. These are located under the `cts/tests/src` directory. These stubs aren't separated into separate test APKs like the tests, so the `cts/tests/src` directory does not have additional top level directories like "widget" or "view." Follow the same principle of putting new classes into a package with a name that correlates to the purpose of your new class. For instance, a stub activity used for testing OpenGL like `GLSurfaceViewStubActivity` belongs in the `android.opengl.cts` package under the `cts/tests/src/android/opengl` directory.

Other Tasks

Besides adding new tests there are other ways to contribute to CTS:

- Fix or remove tests annotated with BrokenTest and KnownFailure.

Submitting Your Changes

Follow the Android Contributors' Workflow to contribute changes to CTS. A reviewer will be assigned to your change, and your change should be reviewed shortly!

Android Compatibility Downloads

Thanks for your interest in Android Compatibility! The links below allow you to access the key documents and information.

Android 2.3

Android 2.3 is the release of the development milestone code-named Gingerbread. Android 2.3 is the current version of Android. Source code for Android 2.3 is found in the 'gingerbread' branch in the open-source tree.

- Android 2.3 Compatibility Definition Document (CDD)
- Android 2.3 R4 Compatibility Test Suite (CTS)

Android 2.2

Android 2.2 is the release of the development milestone code-named FroYo. Source code for Android 2.2 is found in the 'froyo' branch in the open-source tree.

- Android 2.2 Compatibility Definition Document (CDD)
- Android 2.2 R6 Compatibility Test Suite (CTS)

Android 2.1

Android 2.1 is the release of the development milestone code-named Eclair. Source code for Android 2.1 is found in the 'eclair' branch in the open-source tree. Note that for technical reasons, there is no compatibility program for Android 2.0 or 2.0.1, and new devices must use Android 2.1.

- Android 2.1 Compatibility Definition Document (CDD)
- Android 2.1 R5 Compatibility Test Suite (CTS)

Android 1.6

Android 1.6 was the release of the development milestone code-named Donut. Android 1.6 was obsoleted by Android 2.1. Source code for Android 1.6 is found in the 'donut' branch in the open-source tree.

- Android 1.6 Compatibility Definition Document (CDD)
- Android 1.6 R1 Compatibility Test Suite (CTS)

Compatibility Test Suite Manual

The CTS user manual is applicable to any CTS version, but CTS 2.1 R2 and beyond require additional steps to run the accessibility tests.

- Compatibility Test Suite (CTS) User Manual

Older Android Versions

There is no Compatibility Program for older versions of Android, such as Android 1.5 (known in development as Cupcake). New devices intended to be Android compatible must ship with Android 1.6 or later.

Frequently Asked Questions

In This Document

Open Source

What is the Android Open Source Project?

We use the phrase "Android Open Source Project" or "AOSP" to refer to the people, the processes, and the source code that make up Android.

The people oversee the project and develop the actual source code. The processes refer to the tools and procedures we use to manage the development of the software. The net result is the source code that you can use to build cell phone and other devices.

Why did we open the Android source code?

Google started the Android project in response to our own experiences launching mobile apps. We wanted to make sure that there would always be an open platform available for carriers, OEMs, and developers to use to make their innovative ideas a reality. We also wanted to make sure that there was no central point of failure, so that no single industry player could restrict or control the innovations of any other. The single most important goal of the Android Open-Source Project (AOSP) is to make sure that the open-source Android software is implemented as widely and compatibly as possible, to everyone's benefit.

You can find more information on this topic at our Project Philosophy page.

What kind of open-source project is Android?

Google oversees the development of the core Android open-source platform, and works to create robust developer and user communities. For the most part the Android source code is licensed under the permissive Apache Software License 2.0, rather than a "copyleft" license. The main reason for this is because our most important goal is widespread adoption of the software, and we believe that the ASL2.0 license best achieves that goal.

You can find more information on this topic at our Project Philosophy and Licensing pages.

Why is Google in charge of Android?

Launching a software platform is complex. Openness is vital to the long-term success of a platform, since openness is required to attract investment from developers and ensure a level playing field. However, the platform itself must also be a compelling product to end users.

That's why Google has committed the professional engineering resources necessary to ensure that Android is a fully competitive software platform. Google treats the Android project as a full-scale product development operation, and strikes the business deals necessary to make sure that great devices running Android actually make it to market.

By making sure that Android is a success with end users, we help ensure the vitality of Android as a platform, and as an open-source project. After all, who wants the source code to an unsuccessful product?

Google's goal is to ensure a successful ecosystem around Android, but no one is required to participate, of course. We opened the Android source code so anyone can modify and distribute the software to meet their own needs.

What is Google's overall strategy for Android product development?

We focus on releasing great devices into a competitive marketplace, and then incorporate the innovations and enhancements we made into the core platform, as the next version.

In practice, this means that the Android engineering team typically focuses on a small number of "flagship" devices, and develops the next version of the Android software to support those product launches. These flagship devices absorb much of the product risk and blaze a trail for the broad OEM community, who follow up with many more devices that take advantage of the new features. In this way, we make sure that the Android platform evolves according to the actual needs of real-world devices.

How is the Android software developed?

Each platform version of Android (such as 1.5, 1.6, and so on) has a corresponding branch in the open-source tree. At any given moment, the most recent such branch will be considered the "current stable" branch version. This current stable branch is the one that manufacturers port to their devices. This branch is kept suitable for release at all times.

Simultaneously, there is also a "current experimental" branch, which is where speculative contributions, such as large next-generation features, are developed. Bug fixes and other contributions can be included in the current stable branch from the experimental branch as appropriate.

Finally, Google works on the next version of the Android platform in tandem with developing a flagship device. This branch pulls in changes from the experimental and stable branches as appropriate.

You can find more information on this topic at our Branches and Releases.

Why are parts of Android developed in private?

It typically takes over a year to bring a device to market, but of course device manufacturers want to ship the latest software they can. Developers, meanwhile, don't want to have to constantly track new versions of the platform when writing apps. Both groups experience a tension between shipping products, and not wanting to fall behind.

To address this, some parts of the next version of Android including the core platform APIs are developed in a private branch. These APIs constitute the next version of Android. Our aim is to focus attention on the current stable version of the Android source code, while we create the next version of the platform as driven by flagship Android devices. This allows developers and OEMs to focus on a single version without having to track unfinished future work just to keep up. Other parts of the Android system that aren't related to application compatibility are developed in the open, however. It's our intention to move more of these parts to open development over time.

When are source code releases made?

When they are ready. Some parts of Android are developed in the open, so that source code is always available. Other parts are developed first in a private tree, and that source code is released when the next platform version is ready.

In some releases, core platform APIs will be ready far enough in advance that we can push the source code out for an early look in advance of the device's release; however in others, this isn't possible. In all cases, we release the platform source when we feel the version has stabilized enough, and when the development process permits. Releasing the source code is a fairly complex process.

What is involved in releasing the source code for a new Android version?

Releasing the source code for a new version of the Android platform is a significant process. First, the software gets built into a system image for a device, and put through various forms of certification, including government regulatory certification for the regions the phones will be deployed. It also goes through operator testing. This is an important phase of the process, since it helps shake out a lot of software bugs.

Once the release is approved by the regulators and operators, the manufacturer begins mass producing devices, and we turn to releasing the source code.

Simultaneous to mass production the Google team kicks off several efforts to prepare the open source release. These efforts include final API changes and documentation (to reflect any changes that were made during qualification testing, for example), preparing an SDK for the new version, and launching the platform compatibility information.

Also included is a final legal sign-off to release the code into open source. Just as open source contributors are required to sign a Contributors License Agreement attesting to their IP ownership of their contribution, Google too must verify that it is clear to make contributions.

Starting at the time mass production begins, the software release process usually takes around a month, which often roughly places source code releases around the same time that the devices reach users.

How does the AOSP relate to the Android Compatibility Program?

The Android Open-Source Project maintains the Android software, and develops new versions. Since it's open-source, this software can be used for any purpose, including to ship devices that are not compatible with other devices based on the same source.

The function of the Android Compatibility Program is to define a baseline implementation of Android that is compatible with third-party apps written by developers. Devices that are "Android compatible" may participate in the Android ecosystem, including Android Market; devices that don't meet the compatibility requirements exist outside that ecosystem.

In other words, the Android Compatibility Program is how we separate "Android compatible devices" from devices that merely run derivatives of the source code. We welcome all uses of the Android source code, but only Android compatible devices -- as defined and tested by the Android Compatibility Program -- may participate in the Android ecosystem.

How can I contribute to Android?

There are a number of ways you can contribute to Android. You can report bugs, write apps for Android, or contribute source code to the Android Open-Source Project.

There are some limits on the kinds of code contributions we are willing or able to accept. For instance, someone might want to contribute an alternative application API, such as a full C++-based environment. We would decline that contribution, since Android is focused on applications that run in the Dalvik VM. Alternatively, we won't accept contributions such as GPL or LGPL libraries that are incompatible with our licensing goals.

We encourage those interested in contributing source code to contact us via the AOSP Community page prior to beginning any work. You can find more information on this topic at the Getting Involved page.

How do I become an Android committer?

The Android Open Source Project doesn't really have a notion of a "committer". All contributions -- including those authored by Google employees -- go through a web-based system known as "gerrit" that's part of the Android engineering process. This system works in tandem with the git source code management system to cleanly manage source code contributions.

Once submitted, changes need to be accepted by a designated Approver. Approvers are typically Google employees, but the same approvers are responsible for all submissions, regardless of origin.

You can find more information on this topic at the Submitting Patches page.

Compatibility

What does "compatibility" mean?

We define an "Android compatible" device as one that can run any application written by third-party developers using the Android SDK and NDK. We use this as a filter to separate devices that can participate in the Android app ecosystem, and those that cannot. Devices that are properly compatible can seek approval to use the Android trademark. Devices that are not compatible are merely derived from the Android source code and may not use the Android trademark.

In other words, compatibility is a prerequisite to participate in the Android apps ecosystem. Anyone is welcome to use the Android source code, but if the device isn't compatible, it's not considered part of the Android ecosystem.

What is the role of Android Market in compatibility?

Devices that are Android compatible may seek to license the Android Market client software. This allows them to become part of the Android app ecosystem, by allowing users to download developers' apps from a catalog shared by all compatible devices. This option isn't available to devices that aren't compatible.

What kinds of devices can be Android compatible?

The Android software can be ported to a lot of different kinds of devices, including some on which third-party apps won't run properly. The Android Compatibility Definition Document (CDD) spells out the specific device configurations that will be considered compatible.

For example, though the Android source code could be ported to run on a phone that doesn't have a camera, the CDD requires that in order to be compatible, all phones must have a camera. This allows developers to rely on a consistent set of capabilities when writing their apps.

The CDD will evolve over time to reflect market realities. For instance, the 1.6 CDD only allows cell phones, but the 2.1 CDD allows devices to omit telephony hardware, allowing for non-phone devices such as tablet-style music players to be compatible. As we make these changes, we will also augment Android Market to allow developers to retain control over where their apps are available. To continue the telephony example, an app that manages SMS text messages would not be useful on a media player, so Android Market allows the developer to restrict that app exclusively to phone devices.

If my device is compatible, does it automatically have access to Android Market and branding?

Android Market is a service operated by Google. Achieving compatibility is a prerequisite for obtaining access to the Android Market software and branding. Device manufacturers should contact Google to obtain access to Android Market.

If I am not a manufacturer, how can I get Android Market?

Android Market is only licensed to handset manufacturers shipping devices. For questions about specific cases, contact android-partnerships@google.com.

How can I get access to the Google apps for Android, such as Maps?

The Google apps for Android, such as YouTube, Google Maps and Navigation, Gmail, and so on are Google properties that are not part of Android, and are licensed separately. Contact android-partnerships@google.com for inquiries related to those apps.

Is compatibility mandatory?

No. The Android Compatibility Program is optional. Since the Android source code is open, anyone can use it to build any kind of device. However, if a manufacturer wishes to use the Android name with their product, or wants access to Android Market, they must first demonstrate that the device is compatible.

How much does compatibility certification cost?

There is no cost to obtain Android compatibility for a device. The Compatibility Test Suite is open-source and available to anyone to use to test a device.

How long does compatibility take?

The process is automated. The Compatibility Test Suite generates a report that can be provided to Google to verify compatibility. Eventually we intend to provide self-service tools to upload these reports to a public database.

Who determines what will be part of the compatibility definition?

Since Google is responsible for the overall direction of Android as a platform and product, Google maintains the Compatibility Definition Document for each release. We draft the CDD for a new Android version in consultation with a number of OEMs, who provide input on its contents.

How long will each Android version be supported for new devices?

Since Android's code is open-source, we can't prevent someone from using an old version to launch a device. Instead, Google chooses not to license the Android Market client software for use on versions that are considered obsolete. This allows anyone to continue to ship old versions of Android, but those devices won't use the Android name and will exist outside the Android apps ecosystem, just as if they were non-compatible.

Can a device have a different user interface and still be compatible?

The Android Compatibility Program focuses on whether a device can run third-party applications. The user interface components shipped with a device (such as home screen, dialer, color scheme, and so on) does not generally have much effect on third-party apps. As such, device builders are free to customize the user interface as much as they like. The Compatibility Definition Document does restrict the degree to which OEMs may alter the system user interface for areas that do impact third-party apps.

When are compatibility definitions released for new Android versions?

Our goal is to release new versions of Android Compatibility Definition Documents (CDDs) once the corresponding Android platform version has converged enough to permit it. While we can't release a final draft of a CDD for an Android software version before the first flagship device ships with that software, final CDDs will always be released after the first device. However, wherever practical we will make draft versions of CDDs available.

How are device manufacturers' compatibility claims validated?

There is no validation process for Android device compatibility. However, if the device is to include Android Market, Google will typically validate the device for compatibility before agreeing to license the Market client software.

What happens if a device that claims compatibility is later found to have compatibility problems?

Typically, Google's relationships with Android Market licensees allow us to ask them to release updated system images that fix the problems.

Compatibility Test Suite

What is the purpose of the CTS?

The Compatibility Test Suite is a tool used by device manufacturers to help ensure their devices are compatible, and to report test results for validations. The CTS is intended to be run frequently by OEMs throughout the engineering process to catch compatibility issues early.

What kinds of things does the CTS test?

The CTS currently tests that all of the supported Android strong-typed APIs are present and behave correctly. It also tests other non-API system behaviors such as application lifecycle and performance. We plan to add support in future CTS versions to test "soft" APIs such as Intents as well.

Will the CTS reports be made public?

Yes. While not currently implemented, Google intends to provide web-based self-service tools for OEMs to publish CTS reports so that they can be viewed by anyone. CTS reports can be shared as widely as manufacturers prefer.

How is the CTS licensed?

The CTS is licensed under the same Apache Software License 2.0 that the bulk of Android uses.

Does the CTS accept contributions?

Yes please! The Android Open-Source Project accepts contributions to improve the CTS in the same way as for any other component. In fact, improving the coverage and quality of the CTS test cases is one of the best ways to help out Android.

Can anyone use the CTS on existing devices?

The Compatibility Definition Document requires that compatible devices implement the 'adb' debugging utility. This means that any compatible device -- including ones available at retail -- must be able to run the CTS tests.

Contact Us

Thanks for your interest in Android compatibility!

If you have questions about Android compatibility that aren't covered in this site, you can reach us in one of a few different ways. To get the most out of any of these options, please first read "Getting the Most from Our Lists" on the Community page

For General Discussion

The preferred way to reach us is via the compatibility@android.com address.

For CTS Technical Questions

If you have specific issues with the Compatibility Test Suite that require you to disclose information you'd prefer not to be public, you can contact an email address we've set up specifically this purpose: cts@android.com. This email address is for cases that require disclosure of confidential information only, so general questions will be directed back to the public android-compatibility list. Note also that this list is for specific technical questions; general inquiries will also be directed back to the android-compatibility list.

For Business Inquiries

Finally, business inquiries about the compatibility program, including requests to use branding elements and so on, can be sent to the address android-partnerships@google.com. Like the CTS address, this address is for specific, private inquiries; general questions will be directed back to the android-compatibility list.