Table of Contents

Appendices

IX. Building and Customizing Android

Covers getting the Android source, compiling it and adapting it to the architecture of your choice (Intel, ARM, ARM64, MIPS), including building specific components

X. Androidisms in the Kernel

Low level Android idiosyncrasies in the Linux kernel described in detail: Ashmem, Pmem, logging, low memory killer, power management timed GPIO, and the binder.

Introduction
to
Android Architecture

Introduction to Android

Android

- Android is a software stack for mobile devices
  - Not just an operating system, but an entire platform
- Devised by Android Systems, acquired by Google
- Open sourced and made freely available
- Adapted to hundreds of mobile platforms, mostly ARM
- One of many forms of embedded Linux

If you're reading this, you've already no doubt been exposed to Android – one of the most dominant new platforms to have emerged in the last decade. Barely five years old (at the time of writing), it has already made a powerful impact on the mobile world, becoming the operating system of choice for virtually all mobiles, save those of Apple and RIM (Blackberry).

Android was first devised by Android Systems, a startup that was acquired by Google back in 2005. It became known to the public when the **Open Handset Alliance (**a consortium including Google, Broadcom, HTC, LG, Marvell, Nvidia, Sprint, T-Mobile, and others) announced it in late 2007. When ARM joined the consortium, later, it gained widespread adoption – backed by big equipment manufacturers such as Samsung, and HTC, Telcos like T-Mobile and Sprint, and both ARM and NVidia – the leading Chipset manufacturers for mobile devices. Android 1.0 hit the market in late 2008, and has quickly sped past BlackBerry and Symbian, to contend with Apple's iOS for the top spot.

As it is based on Linux, Android remains open source. Due to the Linux kernel license, all kernel changes (modules excluded) must remain open source.

Android can be seen as a form of Embedded Linux. It standardizes an ARM based Linux distribution, but also provides much more – a full operating environment, and rich APIs. Whereas most other embedded Linux distributions, e.g. Montavista, only provided the barebones, in Android developers find a ready-to-use environment with powerful graphic APIs and a full user-mode, java based environment – ensuring them almost device-agnostic portability.

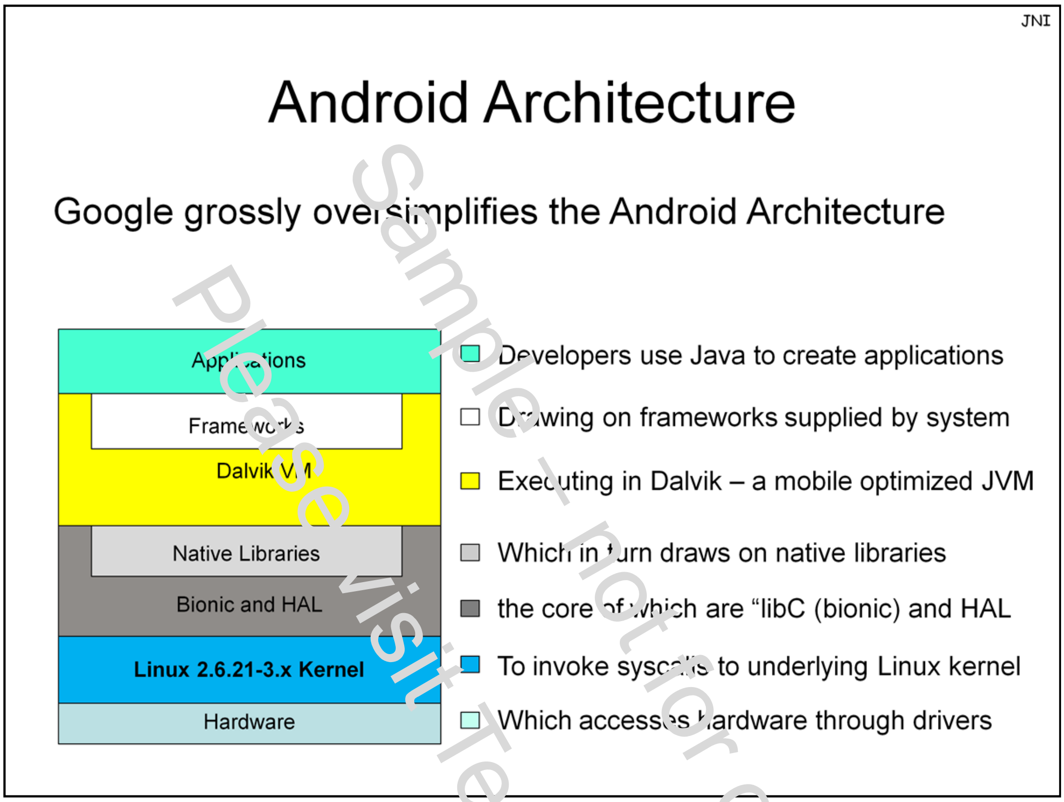
Android – Version History

Version	API	Release	Kernel	Features
1.5 **C**upcake	**3**	4/2009	2.6.27	Widgets, MPEG-4, ..
1.6 **D**onut	**4**	9/2009	2.6.29	Text-To-Speech, speed, gestures
2.0/[.1]/2.1 **É**clair	**5/6/7**	12/2009	2.6.29	Bluetooth 2.1, misc UI
2.2 **F**royo	**8**	5/2010	2.6.32	Speed, V8, JIT, USB Tethering
2.3.0-7 **G**ingerbread	**9/10**	12/2010	2.6.35	Concurrent GC, UI, power mgmt,ext4
3.0 3.1 3.2-3.2.2 **H**oneycomb	**11** **12** **13**	2/2011 5/2011 7/2011	2.6.36	Tablet support, multi-core.. Wi-Fi improvements, Better USB Improved hardware support
4.0- 4.0.3 **I**ce Cream Sandwich	**14** **15**	11/2011 1/2012	3.0.1	Fuse GB + HC: unify tablet/mobile WiFi Direct, Social Stream, NFC
4.1.2-4.2.2 **J**elly Bean	**16-18**	7-11/2012	3.0.31	UI, Search, smoother experience
4.4 Kit Kat	**19**	11/2013	3.4.0	Deep Search, simplified UI, OS. Sensor and Timer batching
?? L	**L**	Late 2014	???	Material Design, Project Volta

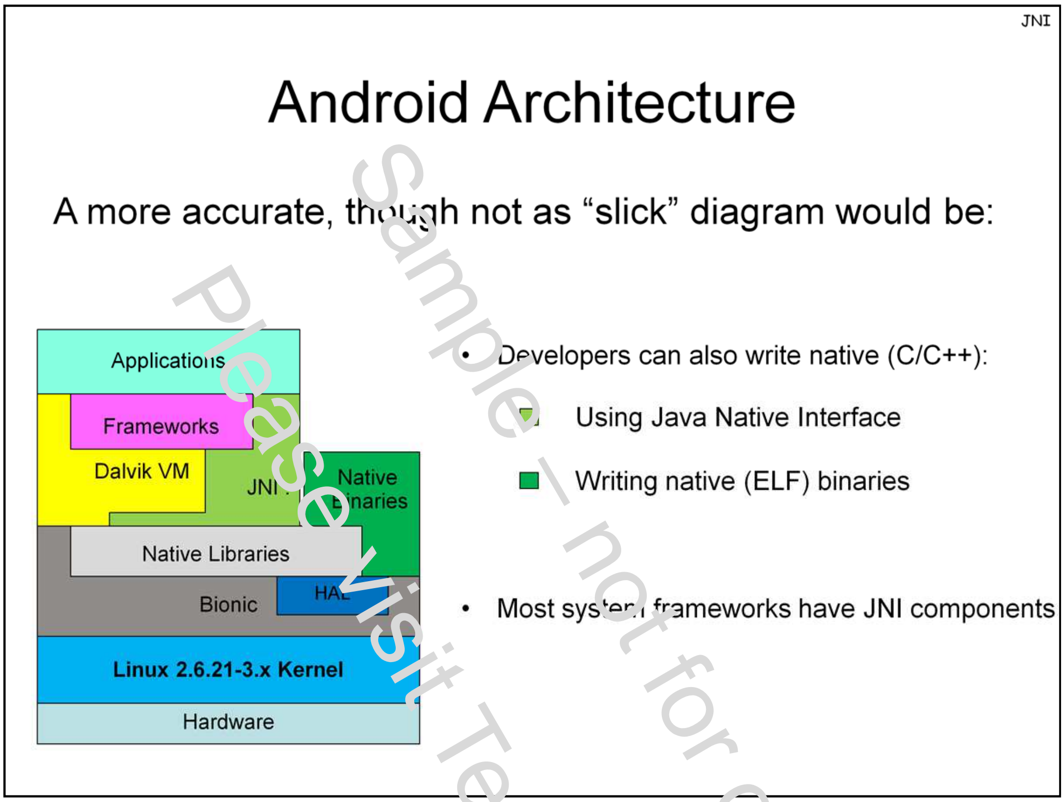
In the few years since it was introduced, Android has gone through a significant number of changes, and many versions. The versions, starting with 1.5, are known by their code names, which are all ordered alphabetically.

The table above lists the versions to date, with the important features they provide. Most of those features are usability and UI features – e.g. exchange connectivity, various codecs and media types, multi-touch interfaces, and others.Most of these features are also provided by the Java based runtime environment. Our scope of discussion, however, will be focused on internal, native features. A full list of features can be found at **http://en.wikipedia.org/wiki/Android_version_history.**

A key concept of Android versioning is that of **API Levels.** API levels are monotonically increasing integer values, starting with 1 (for version 1.0) and currently at 14 (for version 4.0, “Ice Cream Sandwich”). Generally, every version of Android raises the API level by one (with few exceptions, such as versions 2.3.3 and 2.3.4, which held it at 10). This allows an application to declare what API it expects (as part of the manifest, which we discuss next).

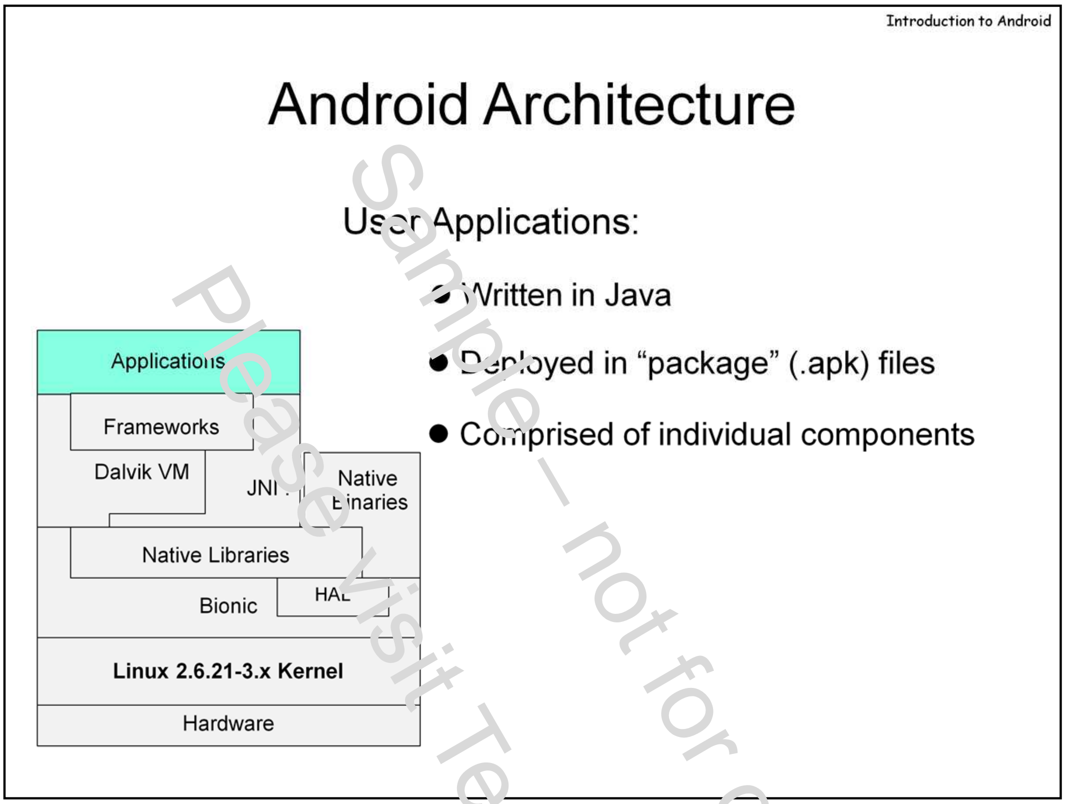


The advertised architecture of Android is overly simplified. Clean, elegant layers, which – like those of iOS – are made simpler and more aesthetic than they are in practice



In practice, however, Google's diagram omits a key component of the architecture - the Java Native Interface (**JNI**), and also entirely ignores native (i.e. Linux ELF) binaries, which make up the foundations of the runtime itself.

We next aim to cover the layers, starting at the Application layer, and moving downward. The discussion, though more detailed than the average, is still only a point of departure, as the rest of this course will delve deeper still into the intricacies and idiosyncrasies.



(Most) Android user applications are written in Java, using the publicly available Android SDK. Using Java enables developers to be relieved of hardware-specific considerations and idiosyncrasies, as well as tap into Java's higher-level language features, such as pre-defined classes.

Applications are comprised of code and resources. Generally, anything that is not code is a resource – this usually means various graphics and configuration files, but also hard coded strings. The code is fully decoupled from its resources, which allows for quick GUI modifcations, as well as internationalization. When deployed, an application is really a single file – a "package" - in a format called .apk. APK is really a modified Java Archive (JAR) file. The file contains the Java classes (in a custom format called .dex – more on that later) which make up the application, as well as an application **manifest**. This concept, which also exists in Microsoft .Net, is of a declarative XML file, which specifies application attributes, required APIs and dependencies, and so forth.

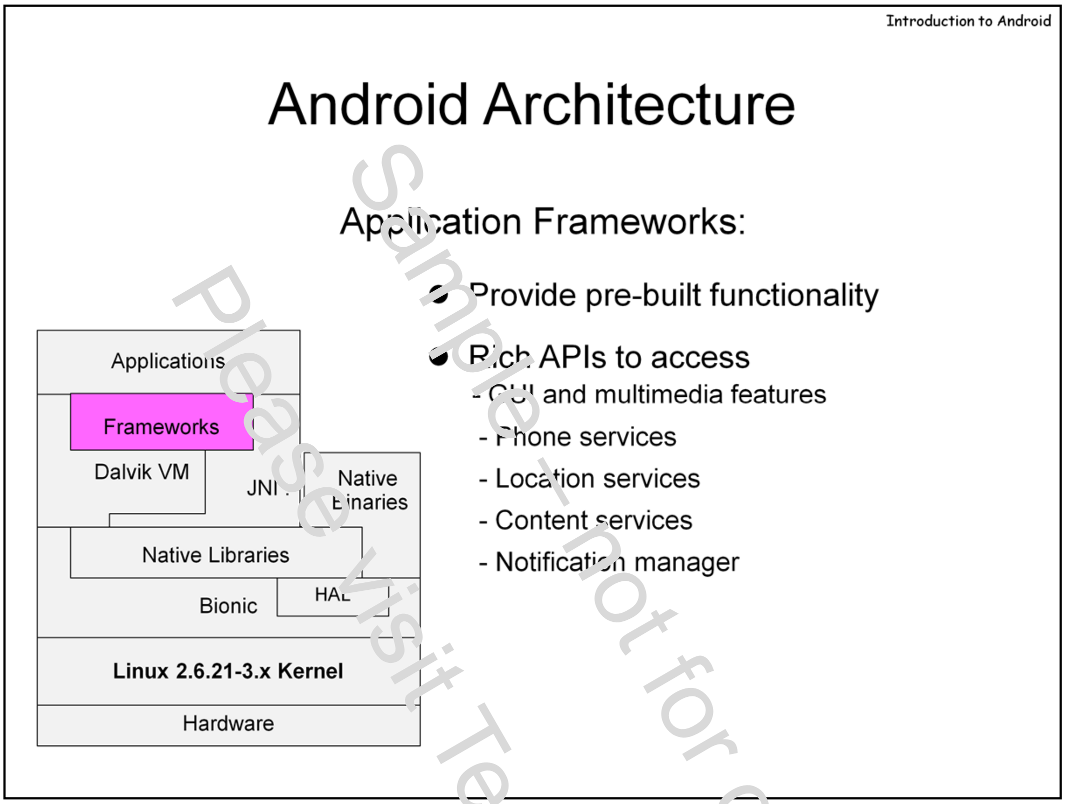
For example, consider the following APK – notice that the standard "jar" utility can be used here. Since .jar itself is .zip compatible, unzip could have done just as well.

```
[root@Forge ~]# jar tvf WidgetPreview.apk
   539 Thu Feb 28 18:33:46 EST 2008 META-INF/MANIFEST.MF
   581 Thu Feb 28 18:33:46 EST 2008 META-INF/CERT.SF
  1714 Thu Feb 28 18:33:46 EST 2008 META-INF/CERT.RSA
  2048 Thu Feb 28 18:33:46 EST 2008 AndroidManifest.xml
 11564 Thu Feb 28 18:33:46 EST 2008 classes.dex
  4773 Thu Feb 28 18:33:46 EST 2008 res/drawable-hdpi/ic_widget_preview.png
  2790 Thu Feb 28 18:33:46 EST 2008 res/drawable-mdpi/ic_widget_preview.png
  1152                              2008 res/layout/activity_main.xml
  2544                              2008 resources.arsc
```

Manifest file (fixed name)

Classes, as a single .dex bundle

Resources (graphics, strings) decoupled from the java code



Application Frameworks are also written in Java, and are based on the low level **core libraries** - which provide the basic subset of Java – java.io.*, java.util.*, etc.

Activity Manager – manages lifecycle of applications. Responsible for starting, stopping and resuming the various applications.

Window Manager – Java abstraction of the underlying surface manager. The surface manager handles the frame buffer interaction and low level drawing, whereas the Window Manager provides a layer on top of it, to allow Applications to declare their client area, and use features like the status bar.

Package Manager – installs/removes applications

Telephony Manager – Allowing interaction with phone, SMS and MMS services

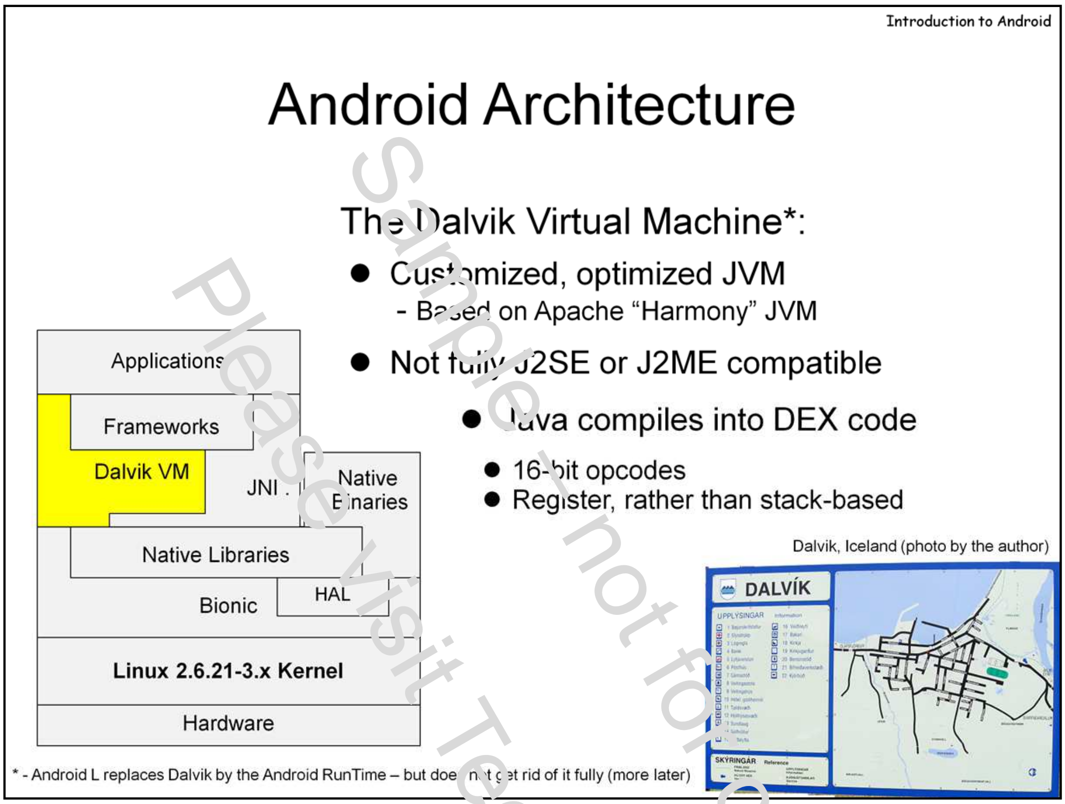
Content Providers – Sharing data between applications – e.g. address book contacts.

Resource Manager – Managing application resources – e.g. localized strings, bitmaps, etc.

View System – Providing the UI primitives - Buttons, listboxes, date pickers, and other controls, as well as UI Events (such as touch and gestures)

Location Manager – Allowing developers to tap into location based services, whether by GPS, cell-tower IDs, or local Wi-Fi databases.

XMPP – Providing standardized messaging (also, Chat) functions between applications



At the heart of Android's user-space lies Dalvik, Android's implementation of the Java Virtual Machine. This is a JVM that has been adapted to the specifics of mobile architectures – systems with limited CPU capabilities (i.e. slow), low RAM and disk space (no swapping), and limited battery life. Under these constraints, the normal JVM – which guzzles memory and is very CPU intensive – would show limited performance.

Enter: Dalvik. Named after a city in northern Iceland, Dalvik is a slimmed down JVM, using less space and executing in those tighter constraints. This Virtual Machine works with its own version of the Java ByteCode, pre-processing its input by using a utility called "dx". This "dx" produces ".dex" (i.e. **D**alvik **EX**ecutable) files from the corresponding Java ".class" files, which are more compact than their counterparts, and offer a richer, 16-bit instruction set. Additionally, Dalvik is a register-based virtual machine, whereas the Sun JVM is a stack-based one. Dalvik instructions work directly on variables (loaded into virtual registers), saving time required to load variables to and from the stack. Register based VMs allow for code that is up to half the size, and runs some 30% faster.

Dalvik code is thus more compact - Even though the instruction size is double that of a normal JVM, .dex files, even when uncompressed, take less space than compressed Java .class files. This is also due to some serious optimizations in strings and method declarations, which enable reuse. Most of the space in Java classes is taken up by constants, which are often repeated, and which Dalvik reduces to one single instance. Dalvik further optimizes code using inline linking, byte swapping, and – as of Android 2.2 – Just-In-Time (JIT) compilation.

It's important to note that Dalvik is neither fully J2SE nor J2ME compatible. For one, due to DEX, classes cannot be created on the fly. Swing and AWT are likewise not supported. The core functionality in Java, however, is supported by Dalvik as well, implemented by the Apache open source "Harmony" JVM implementation.

The user or developer never see .dex – as far as they are concerned – it's all Java. The SDK allows debugging applications with Eclipse as Java files, and the DEX layer is hidden. When deployed, however, it is .dex code that makes it to the device. Dalvik maintains a cache at /data/dalvik-cache:

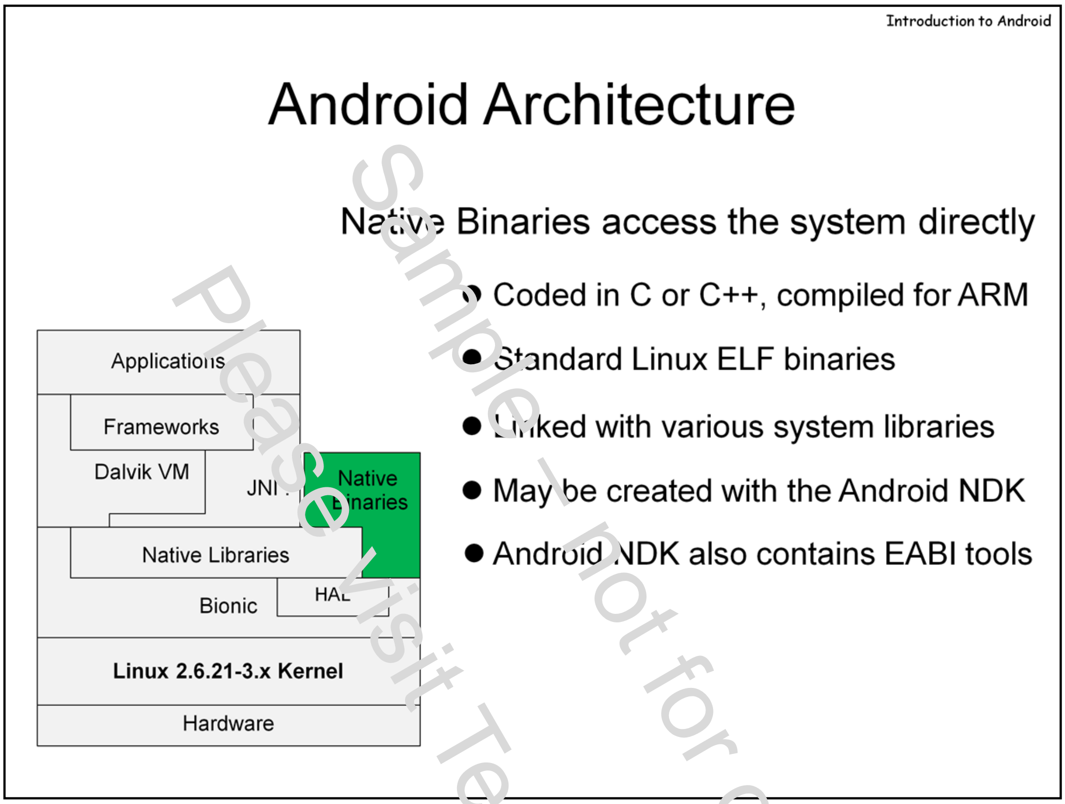
```
root@android:/data/dalvik-cache # ls -s
total 28547
24 system@app@ApplicationsProvider.apk@classes.dex
1359 system@app@Browser.apk@classes.dex
958 system@app@Contacts.apk@classes.dex
625 system@app@ContactsProvider.apk@classes.dex
99 system@app@DeskClock.apk@classes.dex
795 system@app@DownloadProvider.apk@classes.dex
13 system@app@DrmProvider.apk@classes.dex
1279 system@app@Email.apk@classes.dex
900 system@app@Exchange.apk@classes.dex
459 system@app@LatinIME.apk@classes.dex
593 system@app@Launcher2.apk@classes.dex
110 system@app@MediaProvider.apk@classes.dex
712 system@app@Mms.apk@classes.dex
230 system@app@Music.apk@classes.dex
235 system@app@OpenWnn.apk@classes.dex
610 system@app@Phone.apk@classes.dex
1134 system@app@QuickSearchBox.apk@classes.dex
...

root@android# file system\@app\@LatinIME.apk\@classes.dex
system@app@LatinIME.apk@classes.dex: Dalvik dex file
(optimized for host) version 036
```

Normal APK provided classes.dex files are Dalvik version 035 – generic Dalvik. Once deployed on the device itself, they undergo further optimization for processor specific features, which is the "version 0.36" you see, above.

Android contains a tool - /system/xbin/dexdump – which displays very detailed information about dex files, from headers through complete disassembly (q.v. the chapter "Inside an Android"). We'll discuss this tool in detail later on, as well as a replacement (dexter) from Jonathan Levin's "Android Internals" book.

As of KitKat, Dalvik is superseded by the **Android Runtime (ART)**. This, though touted by Google as a replacement, does not in fact replace Dalvik, but merely takes a different approach – Ahead of time compilation (**AOT),** rather than Just-in-Time **(JIT)**. We discuss these differences, as well as both runtime architectures, in great detail later.
.



The Dalvik VM is but one of many **Native Binaries**. These are executables which are compiled directly to the target processor (usually, ARM). Usually coded in C or C++, they can be created with the Android Native Development Kit. The NDK contains a cross compiler, with a full toolchain to create binaries from any platform.

The Android Native binaries are really just standard Linux binaries, and are thus ELF formatted. ELF – the Executable and Library Format – is the default binary format for Linux and most modern UN*X implementations (OS X notwithstanding). The binaries can be inspected using tools like **objdump** and **readelf.**

As an example, consider the following: we begin by using the “adb” command , in the Android SDK, to “pull” (copy to the host) a file from the Android system. In this case, /system/bin/ls. Then we can call “*file*” and “*readelf*” – even those these are running on an x86 host, the ELF file format is still more than readable – revealing that this is really just an ARM-architecture binary:

```
[root@Forge ~]# adb pull /system/bin/ls
398 KB/s (81584 bytes in 0.200s)

[root@Forge ~]# ls -l ls
-rw-r--r-- 1 root root 81584 Jun  8 07:18 ls

[root@Forge ~]# file ls
ls: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses
shared libs), stripped
```



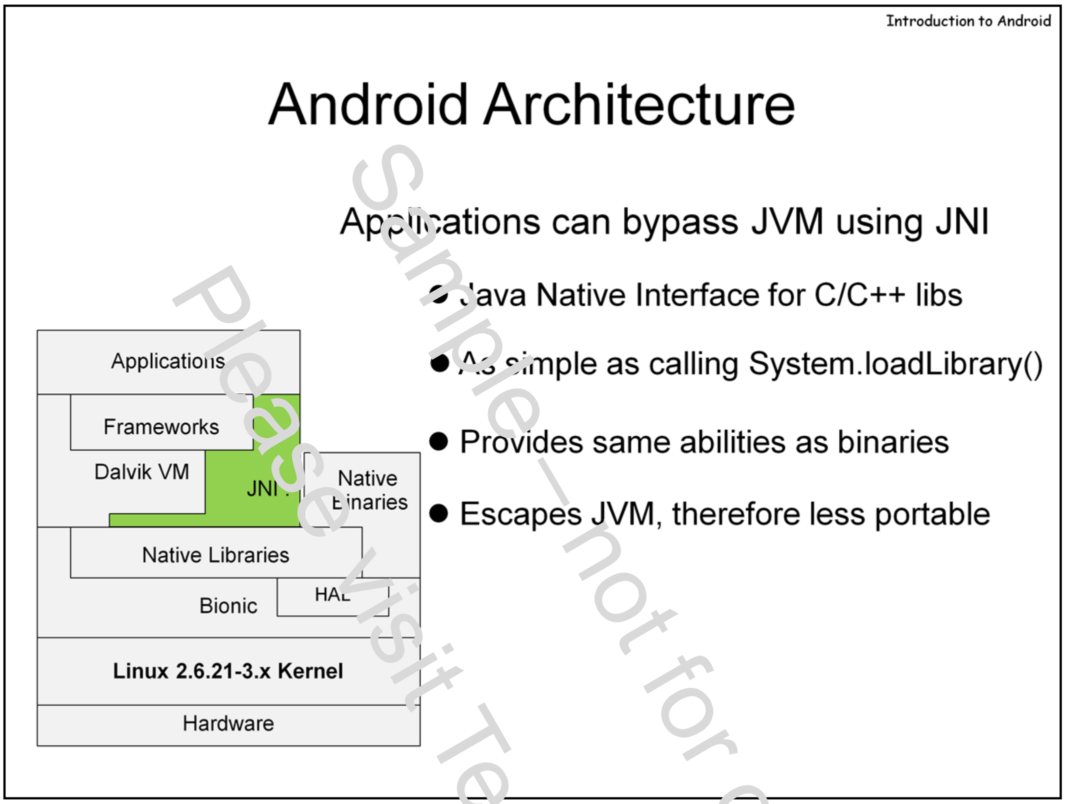
```
[root@Forge ~]# readelf -S ls
There are 25 section headers, starting at offset 0x13ac8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00008114 000114 000013 00   A  0   0  1
  [ 2] .hash             HASH            00008128 000128 000508 04   A  3   0  4
  [ 3] .dynsym           DYNSYM          00008630 000630 000bd0 10   A  4   0  4
  [ 4] .dynstr           STRTAB          00009200 001200 00079b 00   A  0   0  1
  [ 5] .rel.plt          REL             0000999c 00199c 0004f8 08   A  3   2  4
  [ 6] .rel.dyn          REL             00009e94 001e94 000068 08   A  3   2  4
  [ 7] .plt              PROGBITS        00009efc 001efc 000788 00  AX  0   0  4
  [ 8] .text             PROGBITS        0000a690 002690 00be9c 00  AX  0   0 16
  [ 9] .rodata           PROGBITS        0001652c 00e52c 004460 00   A  0   0  4
  [10] .ARM.extab        PROGBITS        0001a98c 01298c 000120 00   A  0   0  4
  [11] .ARM.exidx        ARM_EXIDX       0001aaac 012aac 000420 08   A  8   0  4
  [12] .preinit_array    PREINIT_ARRAY   0001b000 013000 000008 00  WA  0   0  1
  [13] .init_array       INIT_ARRAY      0001b008 013008 000008 00  WA  0   0  1
  [14] .fini_array       FINI_ARRAY      0001b010 013010 000008 00  WA  0   0  1
  [15] .ctors            PROGBITS        0001b018 013018 000008 00  WA  0   0  1
  [16] .data.rel.ro      PROGBITS        0001b020 013020 000558 00  WA  0   0  4
  [17] .dynamic          DYNAMIC         0001b578 013578 0000d8 08  WA  4   0  4
  [18] .got              PROGBITS        0001b650 013650 000314 00  WA  0   0  4
  [19] .data             PROGBITS        0001b964 013964 00000c 00  WA  0   0  4
  [20] .bss              NOBITS          0001b970 013970 005364 00  WA  0   0 16
  [21] .ident            PROGBITS        00000000 013970 000033 00      0   0  1
  [22] .note.gnu.gold-ve NOTE            00000000 0139a4 000018 00      0   0  4
  [23] .ARM.attributes   ARM_ATTRIBUTES  00000000 0139bc 000029 00      0   0  1
  [24] .shstrtab         STRTAB          00000000 0139e5 0000e1 00      0   0  1

Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Tools such as "ldd" in Linux will have issues figuring out dependencies or disassembling the Android binaries. The cross-compiler toolchain tools, however, can work past these difficulties.

```
[root@Forge bin]# pwd
/root/src/android-ndk-r5b/toolchains/arm-eabi-4.4.0/prebuilt/linux-x86/bin
[root@Forge bin]# ls
arm-eabi-addr2line  arm-eabi-g++          arm-eabi-gprof     arm-eabi-readelf
arm-eabi-ar         arm-eabi-gcc          arm-eabi-ld        arm-eabi-run
arm-eabi-as         arm-eabi-gcc-4.4.0    arm-eabi-nm        arm-eabi-size
arm-eabi-c++        arm-eabi-gcov         arm-eabi-objcopy   arm-eabi-strings
arm-eabi-c++filt    arm-eabi-gdb          arm-eabi-objdump   arm-eabi-strip
arm-eabi-cpp        arm-eabi-gdbtui       arm-eabi-ranlib
```

Before we go on to explain the system libraries, it's important to emphasize that application developers can achieve native-level functionality as well, using the **JNI - Java Native Interface**

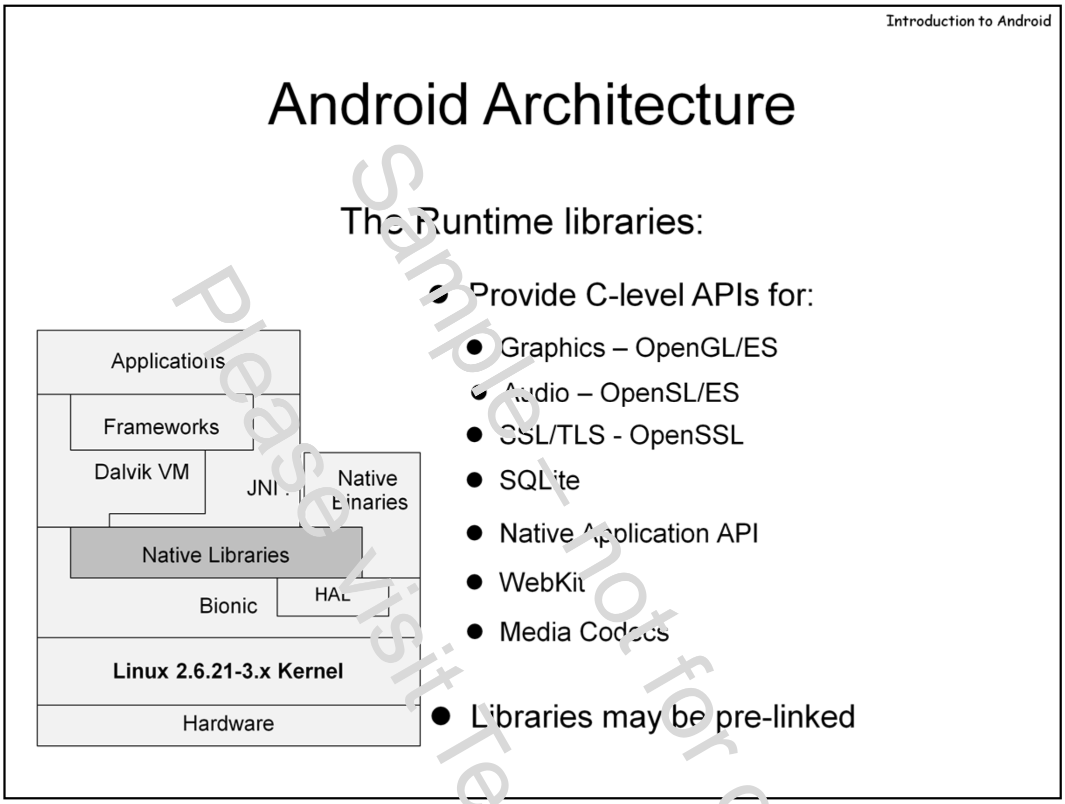
Using JNI enables a Java application to directly invoke a non-Java function, thereby bypassing the JVM, and working on par with native code. Most developers won't ever need to go there, since the runtime environment is so rich – but there are times when a developer might want to access specific hardware functions, such as those of a specialized hardware driver. Doing so is possible, but at the cost of breaking portability.

Good reasons to use JNI are:

- **Efficiency**: For specific applications, such as graphics or high processing applications (e.g. video decoding). JNI can use processor specific features (e.g. ARM NEON), whereas Dalvik usually does not
- **Obfuscation**: Since writing Java code, even when compiling into DEX, is tantamount to open source – anyone can decompile the code very easily - compiling to native code makes it significantly harder to reverse engineer. Code can still be disassembled easily, but that does not offer the same visibility as decompilation does.

The last reason is actually a very important one. Most paid Android app developers opt to use JNI, so that their application isn't easily decompilable. An example is Angry Birds, wherein Rovio places most of the logic inside a "libangrybirds.so", rather than leave it inside the classes.dex.

JNI is discussed in depth in the "Native Binaries" section of this course.



Android provides a rich assortment of runtime libraries. These libraries provide the actual implementation (usually, via system call) of the Android APIs – meaning that when the Dalvik VM wants to execute an operation, it calls on the corresponding library.

The runtime libraries are a collection of many libraries, all open source, which implement the low level functionality provided by the runtime. A full list is maintained as part of the NDK in the STABLE-APIS file.

Library	As of..	Includes	Links with
Bionic (libC)	v1.5	<sys/system_properties> <math.h> <pthread.h>	-lc (default)
DL	v1.5	<dlfcn.h>	-ldl
JNI		<jni.h>	
Logging	v1.5	<android/log.h>	-llog
OpenGL ES 2.0	v2.0	<GLES/gl.h> and <GLES/glext.h>	-lOpenGLES
OpenSL	v2.3	<SLES/OpenSLES.h> <SLES/OpenSLES_Platform.h>	-lOpenSLES
Zlib	v1.5	<zlib.h>	-lz

An important note about libraries,is the prelink feature. Rather than dynamically link needed libraries on binary loading, Android allows for the libraries to be preloaded into memory, so when a process is loaded, it has access to all its libraries (as well as others it might not end up using). This allows for faster load times, and really doesn't waste any memory – as the library code, being text, is all read-only and backed by a single physical copy

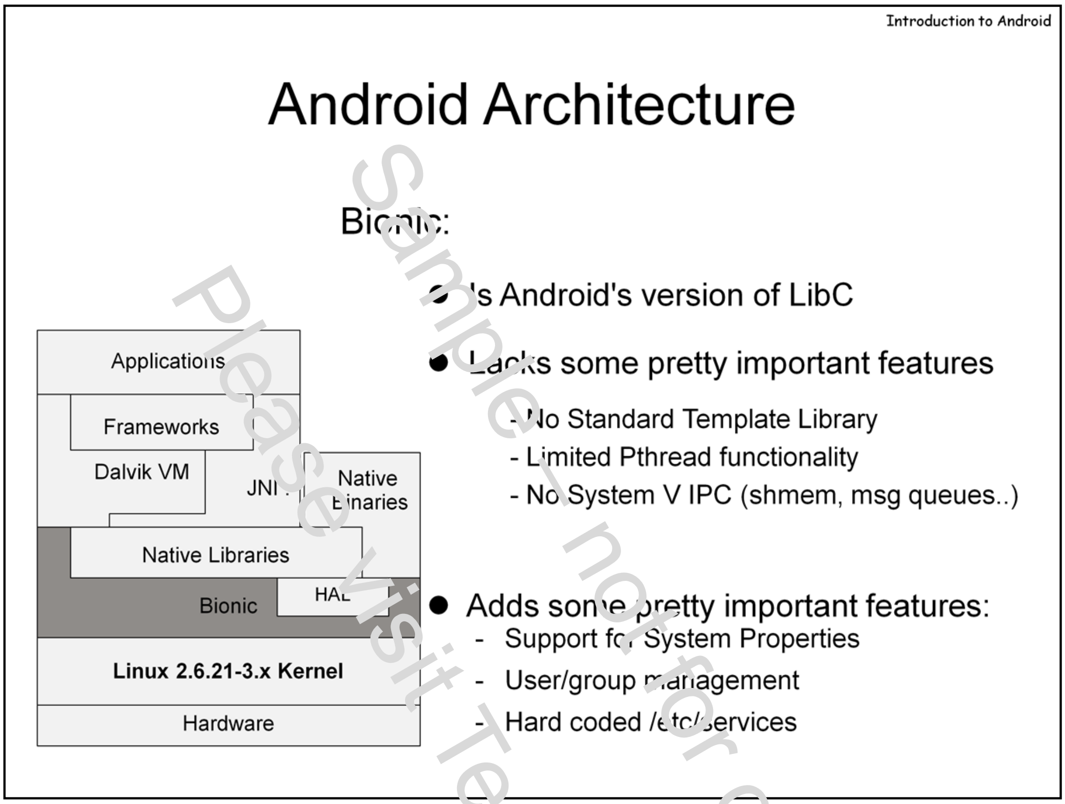
The file maintaining the map is prelink-linux-arm.map, in the build/core directory.

```
# 0xC0000000 - 0xFFFFFFFF Kernel
# 0xB0100000 - 0xBFFFFFFF Thread 0 Stack
# 0xB0000000 - 0xB00FFFFF Linker
# 0xA0000000 - 0xBFFFFFFF Prelinked System Libraries
# 0x90000000 - 0x9FFFFFFF Prelinked App Libraries
# 0x80000000 - 0x8FFFFFFF Non-prelinked Libraries
# 0x40000000 - 0x7FFFFFFF mmap'd stuff
# 0x10000000 - 0x3FFFFFFF Thread Stacks
# 0x00000000 - 0x0FFFFFFF .text / .data / heap

# Note: The general rule is that libraries should be aligned on 1MB
# boundaries. For ease of updating this file, you will find a comment
# on each line, indicating the observed size of the library, which is
# one of:
#
#     [<64K] observed to be less than 64K
#     [~1M] rounded up, one megabyte (similarly for other sizes)
#     [???] no size observed, assumed to be one megabyte
#
# note: look at the LOAD sections in the library header.
#
#   arm-eabi-objdump -x <lib>
#

# core system libraries
libdl.so                0xAFF00000 # [<64K]
libc.so                 0xAFD00000 # [~2M]
libstdc++.so            0xAFC00000 # [<64K]
libm.so                 0xAFB00000 # [~1M]
liblog.so               0xAFA00000 # [<64K]
libcutils.so            0xAF900000 # [~1M]
libthread_db.so         0xAF800000 # [<64K]
libz.so                 0xAF700000 # [~1M]
libevent.so             0xAF600000 # [???]
libssl.so               0xAF400000 # [~2M]
libcrypto.so            0xAF000000 # [~4M]
libsysutils.so          0xAEF00000 # [~1M]

...
```

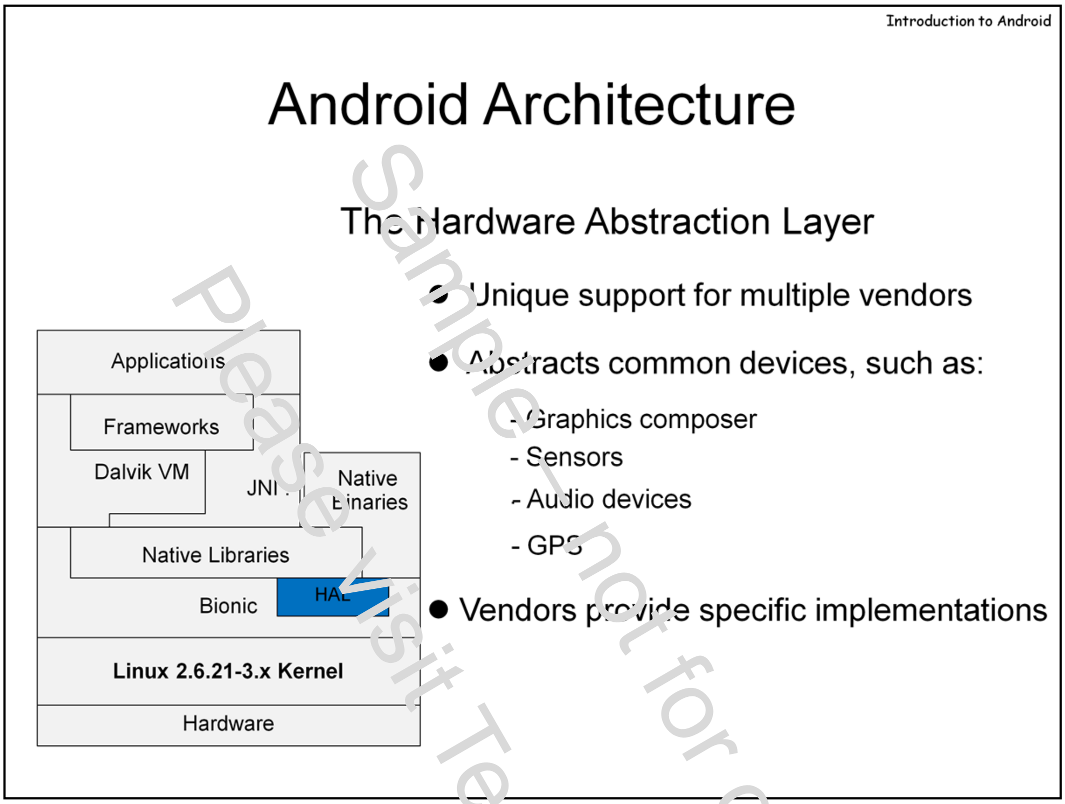
Android uses a custom libC implementation, called **Bionic**. This is a deliberately stripped down version of the standard libC, sacrificing some rarely used features to optimize on memory requirements. Because most of the Applications do not access the library directly – but rather through the Dalvik VM – it made sense to omit them. The list of features added and omitted is part of the source tree, at libc/docs/OVERVIEW.TXT

For example, while Bionic supports threads (a mandatory feature, considering Dalvik threads are backed by Linux threads), the pthread_cancel() API is not supported. Threads can thus not be terminated directly. Another example is the lack of the UN*X standard System V Inter Process Communication (IPC) primitives, such as message queues and shared memory (shmget/shmat/shmdt APIs). Similarly, C++ exception handling is limited. But recall that most of these features aren't required by your average Dalvik based application.

Bionic is now without enhancements, however.:

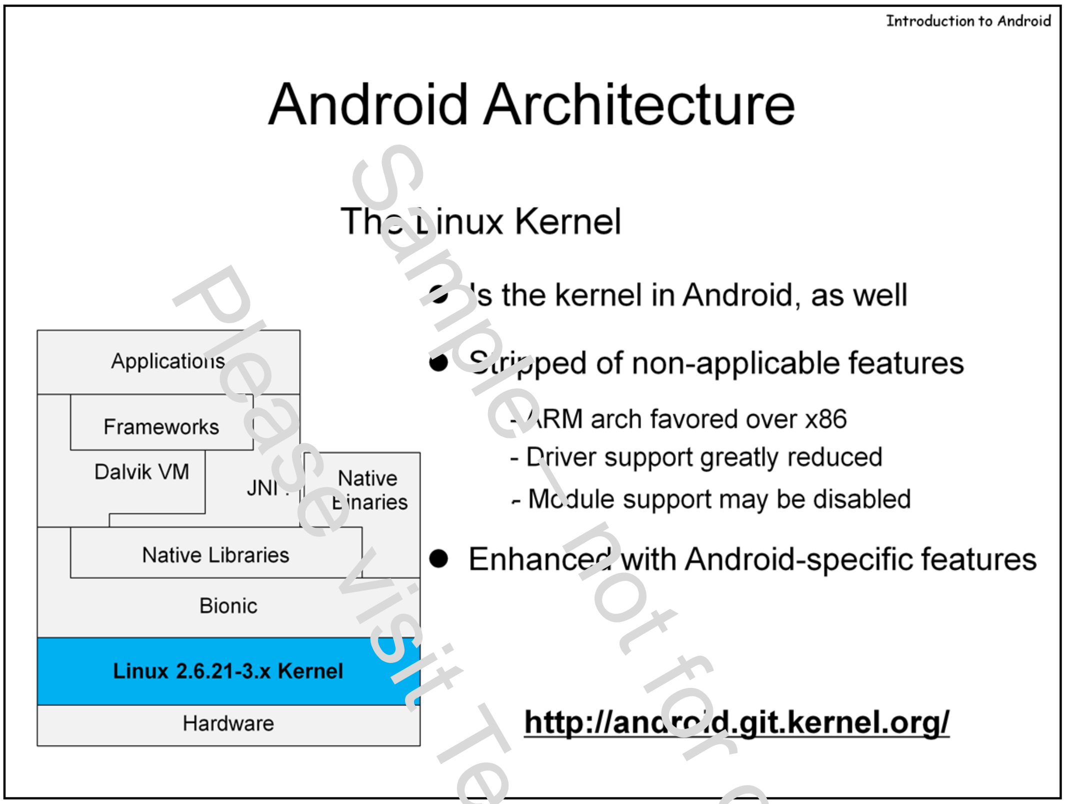
One relatively simple enhancement is support for system wide "properties". These are inherent to Java programming (developers can call System.getProperty or setProperty to query/set JVM parameters, or underlying operating system attributes). They are implemented by system-wide shared memory (started by "init", the user mode process which boots the system), accessible to all processes and, of course, to Dalvik.

Bionic also replaces several /etc functions, most notably /etc/passwd, /etc/group, /etc/services and /etc/nsswitch.conf – none of these files exist on Android, and Bionic provides alternative methods for user/group management, getting service entries, and looking up DNSs (via system properties, or /system/etc/resolv.conf).



A unique feature of Android is the Hardware Abstraction Layer – A special library (libhardware.so) enabling the abstraction of various hardware devices, which would normally be implemented differently by each vendor. The HAL aims to promote standardization by defining an adapter. it only requires the vendor to drop the shim into /system/lib/hw, and the HAL - libhardware.so will automatically load them. For example, this libraries, from a Samsung S5:

```
root@S5:/ # ls -l /system/lib/hw
-rw-r--r-- root     root          9448 2014-03-09 18:21 audio.a2dp.default.so
-rw-r--r-- root     root          5308 2014-03-09 18:21 audio.primary.default.so
-rw-r--r-- root     root        116348 2014-03-09 18:21 audio.primary.msm8974.so
-rw-r--r-- root     root         17708 2014-03-09 18:21 audio.r_submix.default.so
-rw-r--r-- root     root          9476 2014-03-09 18:21 audio.usb.default.so
-rw-r--r-- root     root         13552 2014-03-09 18:21 audio_policy.msm8974.so
-rw-r--r-- root     root       1306732 2014-03-09 18:21 bluetooth.default.so
-rw-r--r-- root     root        280728 2014-03-09 18:21 camera.msm8974.so
-rw-r--r-- root     root          5412 2014-03-09 18:21 consumerir.default.so
-rw-r--r-- root     root         17640 2014-03-09 18:21 copybit.msm8974.so
-rw-r--r-- root     root         26260 2014-03-09 18:21 flp.default.so
-rw-r--r-- root     root         21756 2014-03-09 18:21 gps.default.so
-rw-r--r-- root     root          9736 2014-03-09 18:21 gralloc.default.so
-rw-r--r-- root     root         14328 2014-03-09 18:21 gralloc.msm8974.so
-rw-r--r-- root     root        107820 2014-06-06 13:32 hwcomposer.msm8974.so
-rw-r--r-- root     root          5308 2014-03-09 18:21 keystore.default.so
-rw-r--r-- root     root          5308 2014-03-09 18:21 local_time.default.so
-rw-r--r-- root     root         65412 2014-03-09 18:21 nfc_nci.MSM8974.so
-rw-r--r-- root     root          5316 2014-03-09 18:21 power.default.so
-rw-r--r-- root     root         21924 2014-03-09 18:21 sensorhubs.msm8974.so
-rw-r--r-- root     root         54640 2014-06-06 13:32 sensors.msm8974.so
```

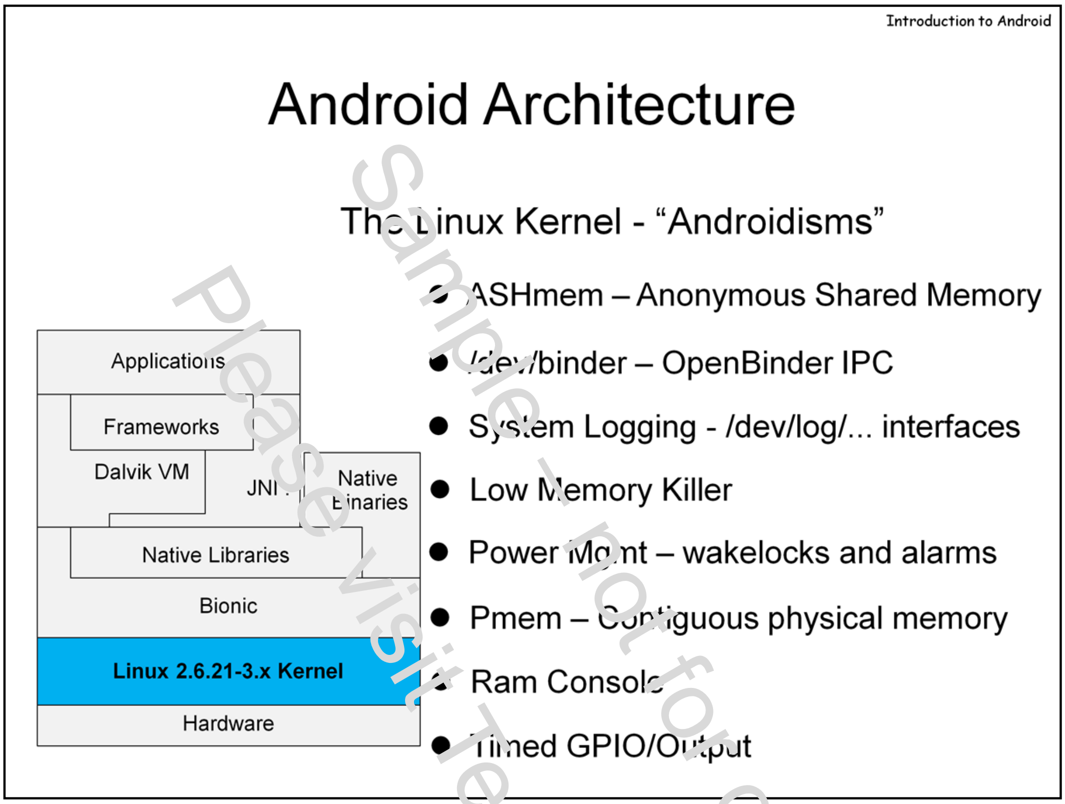
All modern operating systems are based on a **kernel** and Android is no exception. Android uses the open source Linux Kernel as its own, albeit with some (open source) modifications.

For one, the kernel is compiled to mobile architectures. Predominantly, this means ARM instead of the usual Intel (although Intel will surely not be left out of the mobile market for long).

The kernel is similar, though not identical, to the standard Linux kernel distribution, maintained at http://www.kernel.org/. Android strips down many of the drivers which are not applicable in mobile environments, and the default architecture is ARM, rather than x86. Another feature that may be lacking* is module support (though that is a simple #define, when compiling the kernel). The reason for that is making the kernel smaller, and more secure: hardware vendors compile all their drivers into the kernel, and really there is no need for on the fly module loading – which can lead to serious security compromise, by injecting code directly into kernel space.

Although there have been some initiatives to do so, at the time of writing it is unlikely that Android will be merged back into the Linux source tree. There are simply too many changes (and a fair amount of clutter) to incorporate into the main source tree. What more, specific hardware vendors further customize Android still, leading to divergence and excess branching.

* - Depending on how the kernel is built – Module support can easily be toggled in the kernel config.



Android's specific enhancements to the Linux Kernel have been dubbed "Androidisms". These are add-ons to the original kernel source, implementing features which are mobile specific, and generally not as useful or applicable in a desktop or laptop system. Most are all implemented in the /drivers/staging/android part of the source tree, though some – like memory management – are implemented in the corresponding subsystem's directory. The following table lists those features, as well as where to find them in the source tree (if not in drivers/staging/android):

Feature	In	Used for
ashmem	mm/ashmem.c	Anonymous Shared Memory
binder	binder.c	Android's implementation of OpenBinder, and the underlying implementation of the RunTime AIDL
logging	logger.c	Android's enhanced logging, via /dev/log/…. Specific entries
Lowmem killer	lowmemorykiller.c	Layer on top of Linux's "oom" to kill processes when the system is out of memory
Pmem	Drivers/misc/pmem.c	Contiguous physical memory, for systems which need it
RAM console	ram_console.c	Implementation of RAM based physical console (during boot)
Timed GPIO	timed_gpio.c	Timed GP I/O – Manipulate GPIO registers from user space
Timed output	timed_output.c	Timed output

Android has several important **Memory Management extensions,** which the standard kernel does not. The first, **ASHMem,** is a mechanism for anonymous shared memory, which abstracts shared memory as file descriptors. This mechanism, implemented in **mm/ashmem.c,** is used very heavily.

Pmem is a mechanism for allocation of virtual memory that is also physical contiguous. This is required for some hardware, which cannot support virtual memory, or scatter/gather I/O (i.e. access multiple memory regions at once). A good example is the mobile device camera.

The last extension, the **Low Memory Killer**, is built on top of Linux's "OOM" (out-of-memory) mechanism, a feature which was introduced into the Kernel somewhere around 2.6.27(?). This feature is necessary, because remember most mobile devices do not have the luxury of swap – and when the physical memory runs out, the applications using the most of it must be killed. Lowmem enables the system to politely notify the App it needs to free up memory (by means of a callback). If the App cooperates, it lives on. If not, it is killed.

The **binder** is Android's underlying mechanism for IPC. It supports the runtime's "AIDL" mechanism for IPC by means of a kernel provided character device – we discuss this at length later.

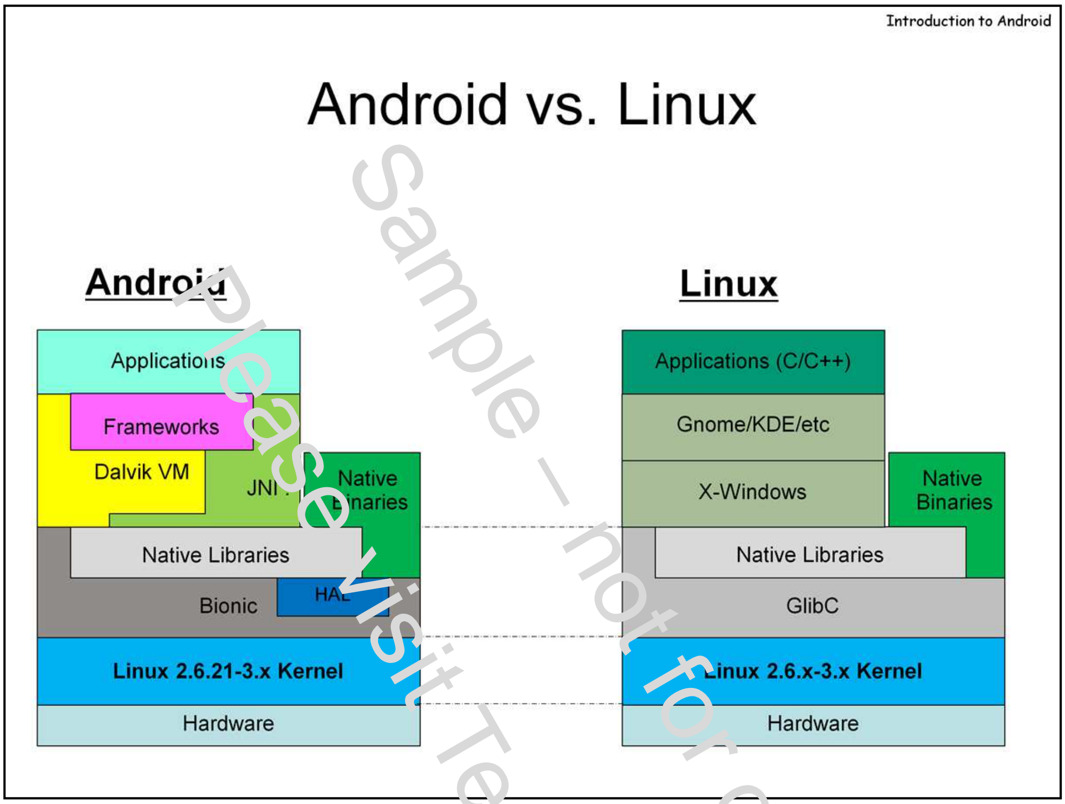
The **logging subsystems** allows separate logfiles for the various subsystems on Android – e.g. radio, events, etc.. The logs are accessible from user mode in the /dev/log directory. On a standard Linux, /dev/log is a socket (owned by syslog). These are really just standard ring buffers, very similar to the standard kernel log, which is present in Android as well, and accessible via the **dmesg** command.

The **RAM Console** is an extension that allows the kernel – when it panics – to dump data to the device's RAM. In a normal Linux, panic data would go right to the swap file – but mobile devices don't have swap (because of Flash lifetime considerations). A RAM Console is a dedicated area in the RAM where the panic data will be stored. Following a panic, the device performs a warm reboot, meaning the RAM is not cleared. When the kernel next boots, this area is checked for the presence of panic data (using a magic value), and – if found – the data is made accessible to user space via the /proc file system (/proc/apanic_console and /proc/apanic_threads). The first user mode process, init, usually collects these files, if they exist, into a persistent store on the file system, /data/dontpanic (an obvious nod to the Hitchhiker's Guide to the Galaxy).

Wakelocks and **alarms** are two **Power management extensions** built into Android. The Linux kernel supports power management, but android adds two new concepts: "**Alarms**" are the underlying implementation of the RunTime's "AlarmManager" - which enables applications to request a timed wake-up service. This has been implemented into the kernel so as to allow an alarm to trigger even if the system is otherwise in sleep mode.

The second concept is that of "**wakelocks**",which enable Android to prevent system sleep. Applications can hold a full or a partial wakelock – the former keeps the system running at full CPU and screen brightness, whereas the latter allows scren dimming, but still prevents system sleep. Though these are kernel objects, they are exported to user space via /sys/power files – wake_lock and wake_unlock, which allow an application to define and toggle a lock by writing to the respective files. A third file, /proc/wakelocks, to show all wakelocks. The runtime wraps these with a higher level Java API using the PowerManager.

We discuss the nooks and crannies of these Android idiosyncrasies later on, in great detail and at the level of the actual source code – in Module VII.



Android is a derivative of Linux, but has gone quite a way to distance itself and, in some ways, become incompatible with its origins. From a high-level perspective, though it's hard to quantify exactly how much the two OSes differ, a safe estimate would be that Android and Linux are about 95% alike at the kernel level (i.e save for "Androidisms"), and about 55% or so at the user-mode (accounting for the frameworks and Dalvik, as well as Bionic). X-Windows and the various desktop environments are no longer – which is a good thing, since the multitude of APIs were hard to work with, and entirely non-standardized.

It's noteworthy that Android draws many of its most powerful features from Linux. It amalgamates different Linux features – cgroups, the device mapper, SELinux and many others – most of which are left unused in desktop distributions – in clever and innovative ways which provide performance, encryption, and security.`

Introduction to Android

Android vs. iOS

The two competing systems are closer than you think:

- Both are UNIX based (Android's Linux vs. iOS's Darwin)
- Both provide frameworks for most functions
- Application deployment is very similar (.apk vs. .ipa)
- Applications implement fixed call-back entry points ("lifecycle")
- Libraries preloaded (Bionic's prelink vs. iOS's dyld cache)

But also light years apart:

- No Java in iOS, at all – everything native, Objective C.
- iOS is sealed tight, Android is wide open

Android's chief adversary in the mobile world is Apple's "iOS". There are as many similarities as there are differences between the two.

Similarities can be found in the way Applications are handled by the operating system. In both cases, applications are archived packages (Android: .apk, iOS: .ipa). Android's apps have "manifest" XML files describing them. In iOS, a similar concept – of property lists – achieves the same functionality.

At the operating system level, both systems are UNIX based. iOS is based on Apple's Darwin (the open source core of Mac OS X), and Android on Linux. Their filesystems are also somewhat similarly structured (though the underlying implementation is different – HFSX in iOS, JFFS or Ext4 in Android). iOS has no Dalvik, though, and the frameworks therein are based on Objective-C – mobile ports of Apple's OS X frameworks.

Differences:

iOS, while based partially on open source (the xnu kernel) remains very much a closed system. This is true for developers (who are expected to program only in user mode using Apple's tools, and cannot modify core system functionality) as well as for its users (who must go to great lengths to "jailbreak" their devices, to allow custom applications and modifications.

iOS apps are compiled to native code, whereas Android apps remain in Java form.

iOS also only works on very specific hardware – Apple's i-Devices (iPhone, iPod, iPad, Apple TV) – all ARM based. Android, by comparison, is as customizable and portable as Linux is.

As you can see from the architectural diagram, below, the functionality of the layers in both OSes is the same – though the implementation is different. For lack of Dalvik, there is no need for a virtual machine – though iOS applications run, for the most part, in the Objective-C runtime. All layers inaccessible to the developer, save for the Application layer.