Cursul #7

Intro & De ce

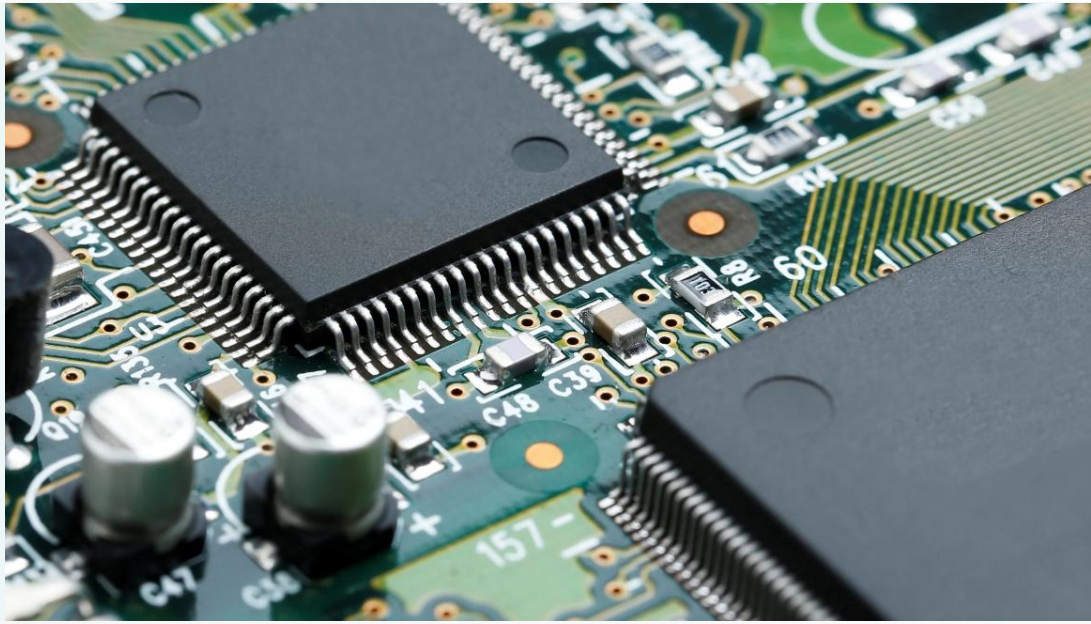
automotive crunch

5/9/2026

SS
automotive crunch

Intro

Android OS



History

- 2003: Started as a startup called Android Inc. in Palo Alto/CA, focused on developing an open operating system for smartphones.
- 2005: Android Inc. acquired by Google.
- 2007: Created the Open Handset Alliance, a consortium of companies interested in the mobile devices area (Google, Intel, TI, Qualcomm, Nvidia, Motorola, HTC, Samsung, etc.).
- 2008: Android version 1.0 is released (HTC Dream).

Versions

- Since then, one or two new versions of Android have been released every year:

Year	Version	Year	Version
2010	2.3 (Gingerbread)	2016	7 (Nougat)
2011	3.0/3.1/3.2 (Honeycomb)	2017	8 (Oreo)
2011	4.0 (Ice Cream Sandwich)	2018	9 (Pie)
2012	4.1/4.2/4.3 (Jelly Bean)	2019	10 (Quince Tart) *
2013	4.4 (KitKat)	2020	11 (Red Velvet Cake)
2014	5 (Lollipop)	2021	12 (Snow Cone)
2015	6 (Marshmallow)	2022	13 (Tiramisu)

- This tutorial is based on version 13.0.

Android in embedded systems

- The Android operating system has surpassed the barrier of mobile devices and is currently used in different embedded system solutions:Smart TVs.
- Vehicle multimedia entertainment centers.
- Cable TV equipment.
- Credit card machines.
- Self-service terminals.
- Smart watches.
- White goods equipment (refrigerators, washing machines, etc.).

Main features

- The operating system base (AOSP) is open source.
- Graphical interface with a very familiar experience.
- Vast application ecosystem (~ 2.7M applications in May/2023). https://www.appbrain.com/stats/number-of-android-apps
- Complete framework for application development, with APIs, IDE (Android Studio), emulator, debugging and testing tools, documentation, books, videos, etc.

Main features (Cont.)

- The base programming language (Java) is familiar and widespread.
- Kotlin, the new official language of Android, is more user-friendly and also quite widespread among developers.
- Full support for Web technologies.
- "Out-of-the-box" hardware support, including:
  - Graphics accelerators via OpenGL ES.
  - Wireless communication technologies (Bluetooth, WiFi, NFC, GSM, CDMA, UMTS, LTE, etc.).
  - Sensors (accelerometer, gyroscope, compass, etc.).

Android open source project

- Android is basically based on two major projects:
  - Linux Kernel (modified).
  - Android Platform (AOSP).
- With each release, Google releases the project's source code through the Android Open Source Project (AOSP).

  https://source.android.com/
- Only a few devices are natively supported by AOSP, including the latest Google devices (Pixel), emulators and some reference boards (DragonBoard 845c, HiKey 960, Khadas VIM3, etc).
  https://source.android.com/docs/setup/create/devices

Community and collaboration

- Several discussion groups are available to communicate with the project developers:

  https://groups.google.com/d/forum/android-platform
- Anyone can contribute to the project through the Gerrit code review tool.

  https://source.android.com/docs/setup/contribute

  https://android-review.googlesource.com
- The evolution of the project and the features that will be available in a future version of Android are controlled exclusively by Google.
  https://code.google.com/p/android/issues/entry

Licenses

- The vast majority of software components are licensed under the permissive Apache and BSD licenses, giving manufacturers the freedom to decide whether they wish to release the modified source code (permissive licenses only require attribution of authorship).
- Some software components are licensed under the GPL/LGPL family of licenses (Google avoids GPL licenses as much as possible).
- Some Google applications are proprietary (Google Play, Gmail, Google Maps, YouTube, etc.).
  - These applications are available in a package called Google Mobile Services (GMS), and to obtain them, it is necessary to certify the device (ACP).

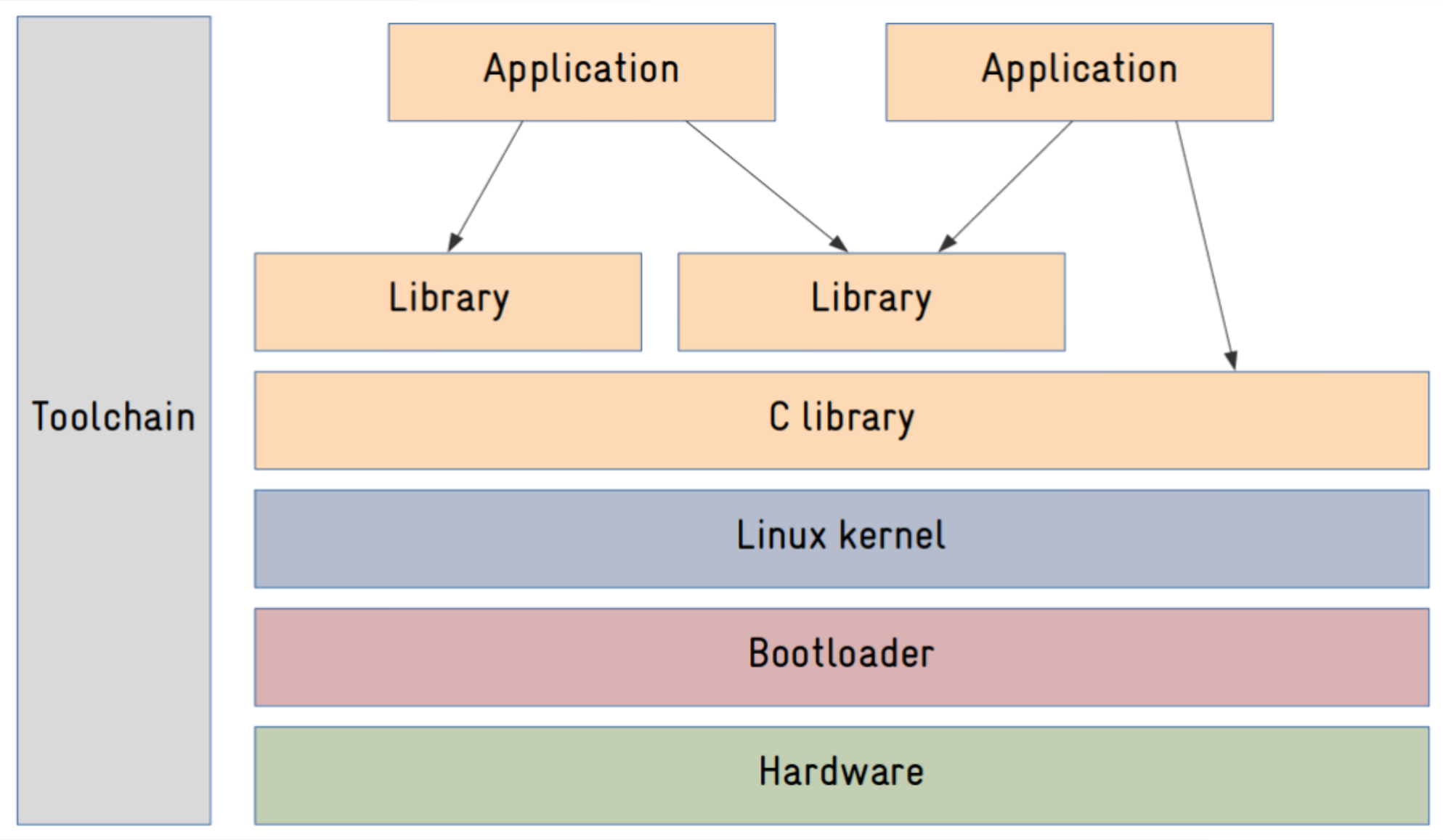
Certification

- In order for a device to be branded as Android and use Google applications, it must be certified through the Android Compatibility Program (ACP):
  - Compatibility Definition Document (CDD): describes the necessary software and hardware requirements for a device to be considered compatible with Android.
  - Compatibility Test Suite (CTS): a tool that provides a set of unit tests to be run on the device to ensure its compatibility with Android.
- More information about the certification process is available on the project website: https://source.android.com/compatibility/overview

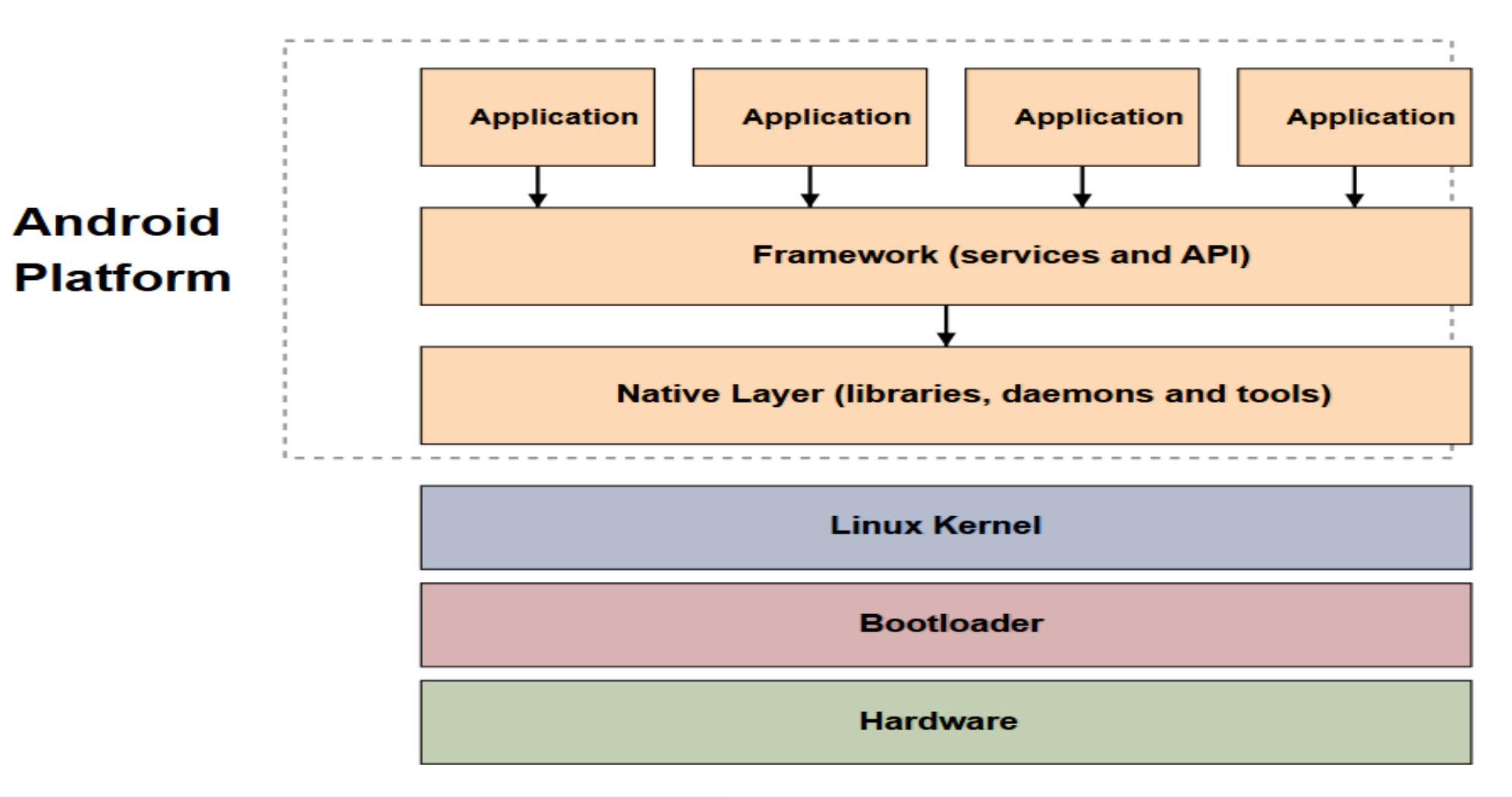
Android variants

- Official Android variants provided by Google:
  - Android Phone.
  - Android Tablet.
  - Android Automotive.
  - Android TV.
  - Android Wear.
- There are also alternative versions of Android developed by the community, including LineageOS (formerly CyanogenMod), CopperheadOS, Replicant, and ProtonAOSP.

GNU/Linux system architecture



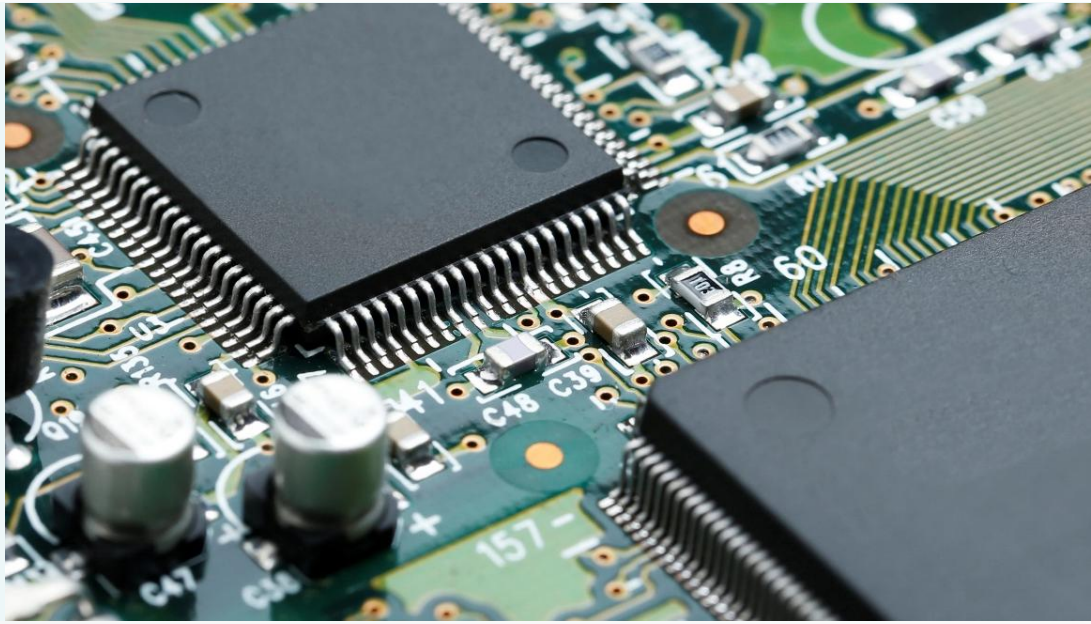
Android architecture



SS
automotive crunch

Android Embedded

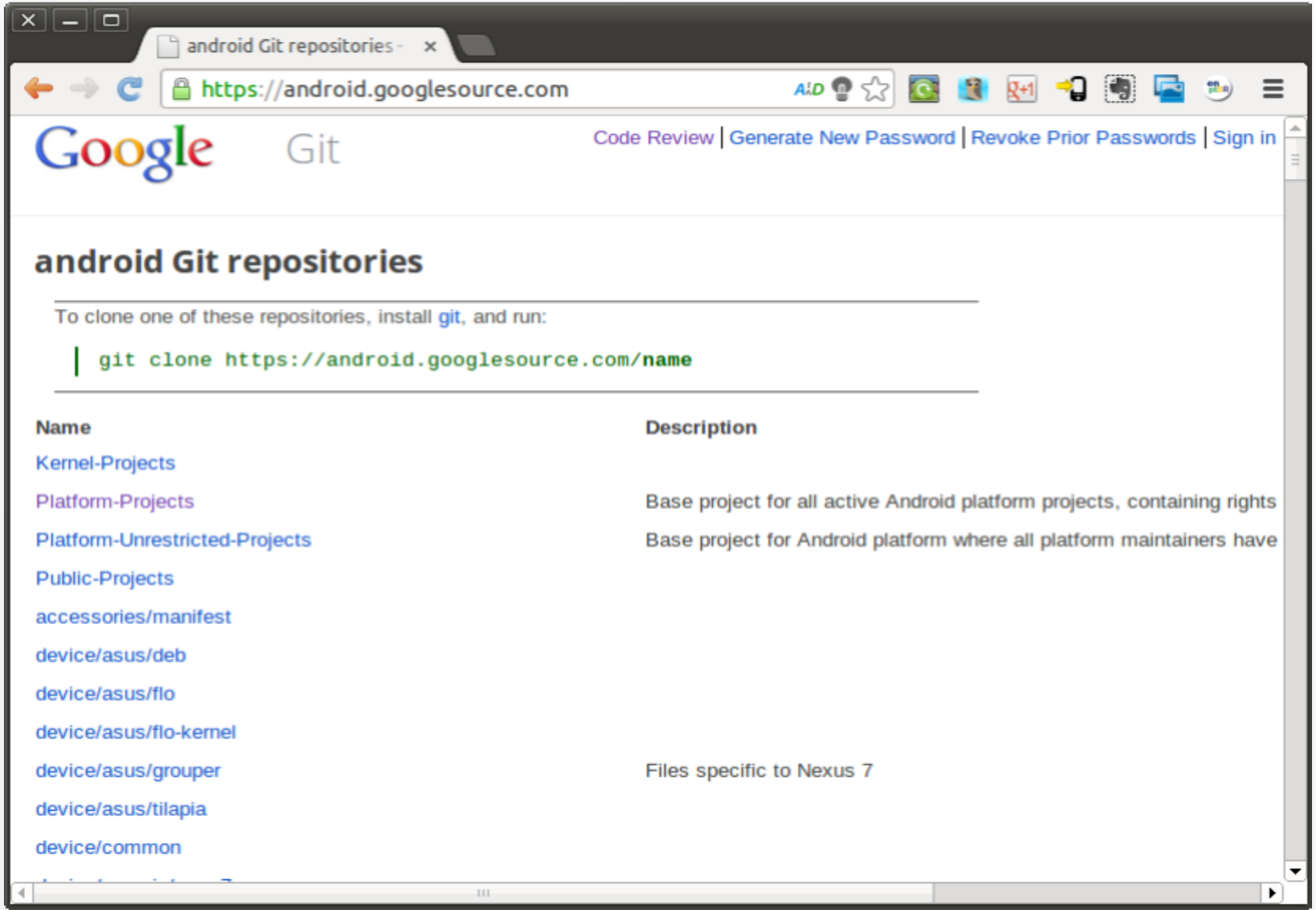
Source code



AOSP and Git repositories

- AOSP is versioned by Google through Git.
- However, the project is divided into hundreds of Git repositories.
  - If the project were managed by just one Git repository it would be slow to download and difficult to manage!
- The Android Git repositories can be accessed at the link below: https://android.googlesource.com/

ANDROID.GOOGLESOURCE.COM



Repo tool

- To manage the hundreds of Git repositories that exist in AOSP, Google created a tool called repo. https://android.googlesource.com/tools/repo
- The repo can be installed according to the instructions in the link below: https://source.android.com/docs/setup/download#installing-repo
- Through a manifest file that describes all the repositories that make up the project (manifest.xml), the repo tool is able to download and manage a specific version of AOSP.

Manifest.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>

  <remote  name="aosp"
           fetch=".."
           review="https://android-review.googlesource.com/" />
  <default revision="refs/tags/android-13.0.0_r44"
           remote="aosp"
           sync-j="4" />

  <superproject name="platform/superproject" remote="aosp" revision="android-13.0.0_r44" />
  <contactinfo bugurl="go/repo-bug" />

  <project path="build/make" name="platform/build" groups="pdk" >
  <project path="build/bazel" name="platform/build/bazel" groups="pdk" >
  <project path="build/blueprint" name="platform/build/blueprint" groups="pdk,tradefed" />
  <project path="build/pesto" name="platform/build/pesto" groups="pdk" />
  <project path="build/soong" name="platform/build/soong" groups="pdk,tradefed" >
  <project path="art" name="platform/art" groups="pdk" />
  <project path="bionic" name="platform/bionic" groups="pdk" />
  <project path="bootable/recovery" name="platform/bootable/recovery" groups="pdk" />
  <project path="cts" name="platform/cts" groups="cts,pdk-cw-fs,pdk-fs" />
  ...
```

Download AOSP with Repo

```
$ repo init -u https://android.googlesource.com/platform/manifest

$ repo sync -j8

$ ls
Android.bp          cts                 hardware            prebuilts
art                 dalvik              kernel              sdk
bionic              developers          libcore             system
bootable            development         libnativehelper     test
bootstrap.bash      device              packages            toolchain
build               external            pdk                 tools
BUILD               frameworks          platform_testing    WORKSPACE
```

Repo subcommands

- *repo diff*: performs a diff on all Git repositories.
- *repo status*: Checks the status of all Git repositories.
- *repo start*: creates a new branch in AOSP.
- *repo branches*: displays existing branches.
- *repo forall*: Runs a command on all Git repositories.
- *repo help*: displays a full menu of options.

Repo and Collaboration

- Google has developed a tool called Gerrit to facilitate the process of collaboration and code review. https://android-review.googlesource.com
- Through the git and repo tools, anyone can contribute to Android development:

```
$ repo start <nome_do_branch>


$ git add -A
$ git commit -s


$ repo upload
```

AOSP Forks

- AOSP is the official Android repository, but it offers limited support for hardware devices.
- Development kit manufacturers can maintain manifest files specific to their hardware platforms:

https://github.com/technexion-android/manifest

https://github.com/hardkernel/android

https://github.com/raspberry-vanilla/android_local_manifest

- The community can maintain specific forks of Android focused on consumer devices (smartphones and tablets):

https://github.com/LineageOS/android

https://github.com/ProtonAOSP/android_manifest

AOSP Forks

- SoC (System-on-Chip) manufacturers such as NXP and Qualcomm also make available to their customers a modified version of AOSP for their processors:

Android OS for i.MX Applications Processors

https://developer.qualcomm.com/hardware/dragonboard-410c/software

- It is also possible to find some hardware manufacturers that make available the entire AOSP source code tree through a compressed file!

Directories: Hardware

- bootable/: reference image of the recovery partition.
- kernel/: configuration files, precompiled images and test scripts of the kernel.
- hardware/: definition of HAL interfaces and reference implementations for some hardware devices.
- device/: product configurations.

Directories: Native layer

- bionic/: Android standard C library.
- libnativehelper/: JNI interface library.
- system/: native layer applications and libraries.
- external/: external projects used by Android (libusb, sqlite, mksh, etc.).

Directories: Framework and applications

- dalvik/: Dalvik virtual machine.
- art/: ART virtual machine.
- libcore/: Java standard library (fork of the Apache Harmony project).
- frameworks/: framework source code (APIs and services).
- packages/: Android applications.

Directories: Tools

- sdk/: SDK (So ware Development Kit) tools.
- pdk/: PDK (Platform Development Kit) tools.
- development/, test/, platform_testing/, developers/, tools/: development, testing and debugging tools.
- cts/: CTS (Compatibility Test Suite) tools.

Directories: Build system

- build/: Android build scripts and files.
- prebuilts/: Prebuilt binaries, including the toolchain (based on Clang).
- toolchain/: Toolchain test and benchmark scripts.

Navigating the source code

- As there is not much documentation regarding AOSP, it is common to need to browse the source code to understand a certain Android functionality.
- To do this, we can generate an AOSP project file and use Android Studio to browse the Android source code.

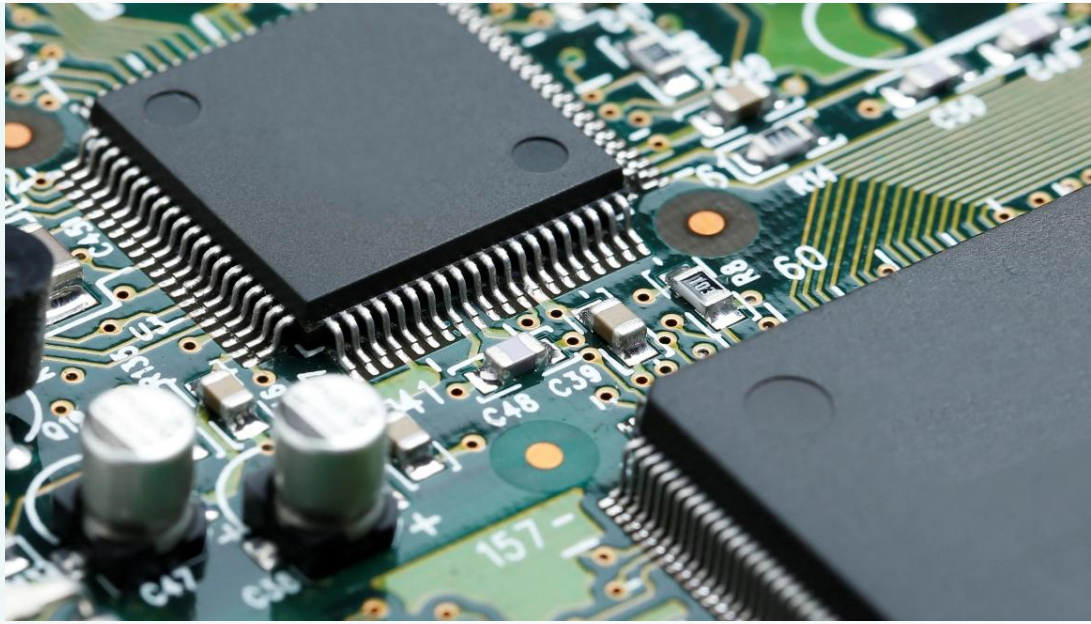
```
$ cat development/tools/idegen/README
$ cat tools/asuite/aidegen/README.md
```

- Other ways to browse the official Android source code include Google Code Search and AOSPXRef

https://cs.android.com

https://aosp.opersys.com

http://aospxref.com



Build system

Build systems

- Build systems have the main purpose of integrating all the software components of a Linux system (toolchain, bootloader, kernel, libraries and applications) and building the operating system image.
- In Linux, the two main build system projects are:
  - Buildroot.
  - Yocto Project/OpenEmbedded.
- However, Android has its own build system!

AOSP build system

- Until Android 6 (Marshmallow), the Android build system was purely based on makefiles, processed by the make tool.
  - The compilation instructions for all software components were defined in Android.mk files.
- This build system had several shortcomings, including instability, low performance, and difficulty in tracking and debugging its execution.
- It was replaced in Android 7 (Nougat) by the Soong build system.

https://source.android.com/setup/build#what_is_soong/

https://android.googlesource.com/platform/build/soong

Soong build system

- In the Soong build system, the rules for compiling the software components are defined in Blueprint files (Android.bp), which have a syntax similar to JSON.
- The Blueprint files are processed and transformed into files with the .ninja extension, which have all the rules for processing the operating system's software components.
- The files with the .ninja extension are processed by the Ninja tool, which is responsible for executing the compilation commands, generating the software artifacts, and building the operating system images.

https://ninja-build.org/

Migration to Soong

- Until Android 13, not all make files (Android.mk) were converted toBlueprint files (Android.bp).

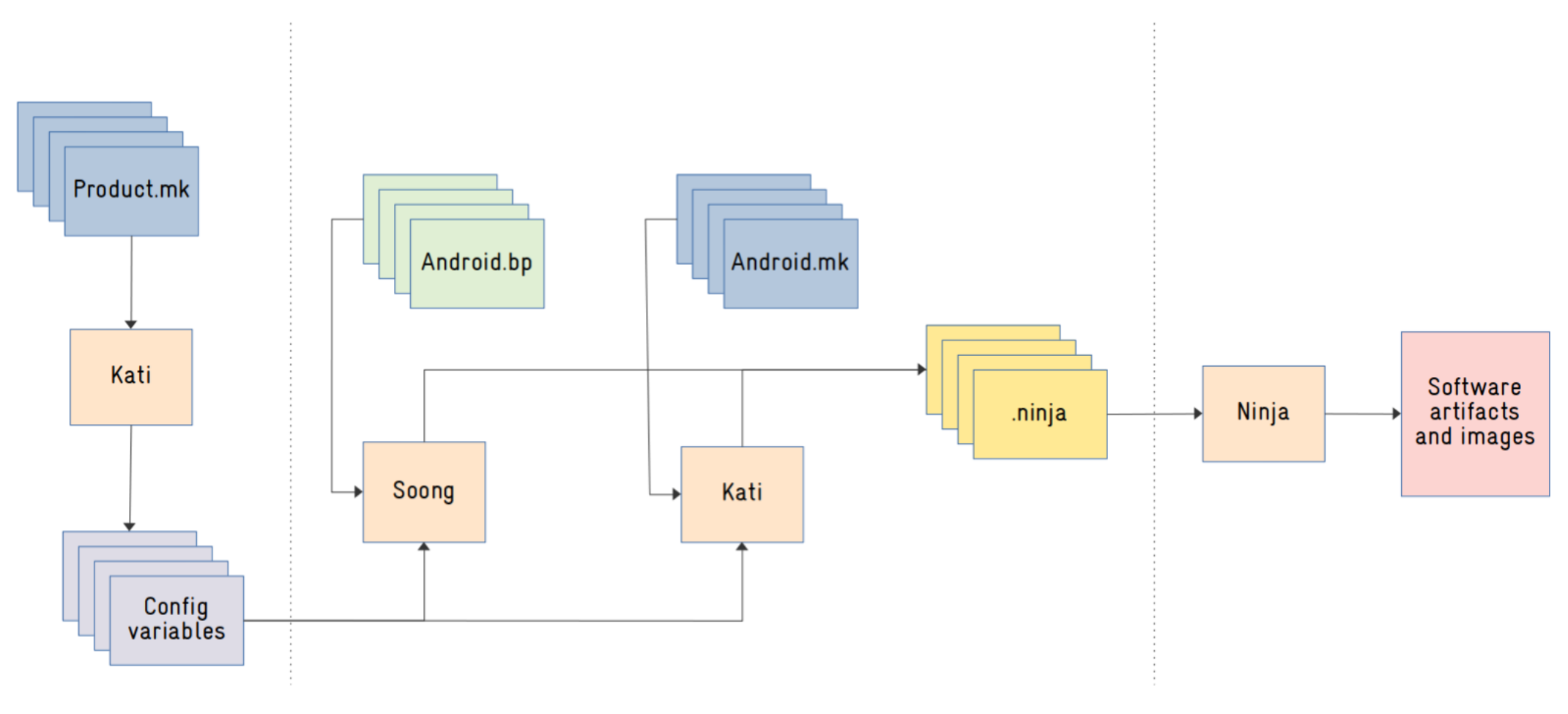
```
$ find . -name Android.bp | wc -l
8857
$ find . -name Android.mk | wc -l
967
```

- For this reason, Google created a tool called kati to convertAndroid.mk files to .ninja files.

https://github.com/google/kati

- Android.mk files are gradually being converted to Android.bp, which will eliminate the need for makefiles and the kati tool in AOSP.

Diagram: Soong



Bazel

- Bazel is an open source build system maintained by Google and used extensively in its software projects. https://bazel.build
- There is a way to convert the AOSP build system to Bazel.
  - Product makefiles are converted to Starlark (*.blz) format.
  - Blueprint files are converted to Bazel build files (BUILD).
  - Soong and ninja are replaced by Bazel for the build process.
- See a presentation from the Android team for more details: Long-term initiative https://www.youtube.com/watch?v=VyW04BwiBSs

Prerequisites for compiling AOSP

- A 64-bit machine with Ubuntu is recommended (configuration used and tested by Google), but the build system should work on other Linux distributions.
- A machine with good processing power (8+ CPUs), plenty of memory (16GB+ RAM) and disk space (256GB+ for a full build, SSD recommended) is required.

https://source.android.com/docs/setup/start/requirements#hardware-requirements

- Several software prerequisites: git, python 3, make, unzip, curl, libncurses, etc.
- More information about building environment setup:https://source.android.com/setup/build/initializing

Initializing the build environment

- The first step to using the Android build system is to load the environment setup script (build/envsetup.sh):

```
$ source build/envsetup.sh
```

- This script will change the configuration of the current terminal, defining some commands, environment variables, and macros that will be used during the compilation process.
- The source command is necessary to ensure that the changes are applied to the user's current terminal.

Envsetup.sh

- Here are some of the commands created by envsetup.sh:
  - lunch: selects the product and build variant for compilation.
  - godir: goes to the directory containing the specified file.
  - gomod: goes to the directory of a module.
  - croot: goes back to the AOSP root directory (or a subdirectory from the root).
  - cgrep: performs a search on .c, .cpp, and .h files.
  - jgrep: performs a search on .java files.
  - hmm: displays the complete list of available commands.

Product and variant

- After running the envsetup.sh script, the next step is to configure the product and the build variant that you want to compile.
- A product defines the hardware and software characteristics that will be used to generate the operating system image.
- The build variant indicates the type of compilation you want to do:
  - user: production build.
  - userdebug: production build with root access and debugging tools.
  - eng: development build with root access and various testing, development, and debugging tools.

Configuring the product

- Product and build variant configuration is performed through environment variables, including:
  - TARGET_PRODUCT: product name.
  - TARGET_BUILD_VARIANT: build variant.
- These variables can be set manually or in a file called buildspec.mk, which should be stored in the AOSP root directory (see example in build/buildspec.mk.default).
- In any case, the most common mechanism for configuring the product and build variant is through the lunch command.

Lunch

- The lunch command will display a list of options in the format <product>-<variant> (also called a combo), and the user must select one of these options.
- It is also possible to run the lunch command by passing the combo directly:

  $ lunch aosp_cf_x86_phone-eng
- When selecting the combo, the lunch command will create the environment variables necessary for compilation:

  $ env | grep "^ANDROID\|^TARGET“
  ANDROID_BUILD_TOP=/opt/build/android
  TARGET_PRODUCT=aosp_cf_x86_phone
  TARGET_BUILD_VARIANT=eng
  …

Compiling

- After selecting the product, Android can be compiled with the m command (it works inside any AOSP directory): $ m
- The -j parameter can be used to parallelize the compilation according to the number of CPUs on the development machine. $ m -j4
- If only -j is passed, the build system will automatically select the number of compilation threads. $ m -j

Compilation output

- The entire Android compilation process takes place in the out/ directory:
  - The tools, executables, and libraries compiled for the host are installed in the out/host/ directory.
  - The software artifacts generated for the target are installed in the out/target/ directory.
- The final images will be available in the product output directory at out/target/product/<product_name>/.

Final images

```
$ ls out/target/product/vsoc_x86/*.img
out/target/product/vsoc_x86/boot.img
out/target/product/vsoc_x86/dtb.img
out/target/product/vsoc_x86/init_boot.img
out/target/product/vsoc_x86/metadata.img
out/target/product/vsoc_x86/misc.img
out/target/product/vsoc_x86/odm_dlkm.img
out/target/product/vsoc_x86/odm.img
out/target/product/vsoc_x86/product.img
out/target/product/vsoc_x86/ramdisk.img
out/target/product/vsoc_x86/super_empty.img
out/target/product/vsoc_x86/super.img
out/target/product/vsoc_x86/system_dlkm.img
out/target/product/vsoc_x86/system_ext.img
out/target/product/vsoc_x86/system.img
out/target/product/vsoc_x86/system_other.img
out/target/product/vsoc_x86/userdata.img
out/target/product/vsoc_x86/vbmeta.img
out/target/product/vsoc_x86/vbmeta_system.img
out/target/product/vsoc_x86/vendor-bootconfig.img
out/target/product/vsoc_x86/vendor_boot-debug.img
out/target/product/vsoc_x86/vendor_boot.img
out/target/product/vsoc_x86/vendor_boot-test-harness.img
out/target/product/vsoc_x86/vendor_dlkm.img
out/target/product/vsoc_x86/vendor.img
out/target/product/vsoc_x86/vendor_ramdisk-debug.img
out/target/product/vsoc_x86/vendor_ramdisk.img
out/target/product/vsoc_x86/vendor_ramdisk-test-harness
```

- The m command has several compilation targets that can be displayed with the help option:

  $ m help

- Some interesting targets:
  - droid: default target (when no target is passed).
  - clean: removes the compilation output directory (rm -rf out/).
  - snod: regenerates the system partition image.
  - vnod: regenerates the vendor partition image.
  - pnod: regenerates the product partition image.

Compiling specific modules

- The m command is capable of compiling specific modules of the build system:

  $ m logcat

- Modules within a specific directory of the build system can be compiled with the mm command:

  $ cd system/logging/logcat

  $ mm logcat

- The same can be done with the mmm command, passing the source code directory of the module you want to compile:

  $ mmm system/logging/logcat:logcat

Regenerating images

- If you have changed an application or product settings, simply use the m command to recompile and regenerate the Android images:

  ```
  $ m -j
  ```

- Alternatively, you can recompile only the changed module and regenerate the corresponding image (in some cases, this method may be faster):

  ```
  $ m logcat
  $ m snod
  ```

Quick testing of changes

- If you have changed the code of an application and want to do a quick test, instead of recompiling Android and regenerating the images, you can recompile just the application and transfer it to the device with ADB:

```
$ m logcat
$ adb remount
$ adb sync
```

- The 'adb remount' command can be replaced by syswrite.
- This procedure is much faster compared to recompiling and regenerating Android images, but it may not work in some situations such as changes to system boot files or product configuration.

Additional resources

- Basic documentation for the m command is available in the AOSP source code at.
- After a build, more complete information about the build process will be available in a log file and can be extracted with the command below:

  ```
  $ gzip -cd out/verbose.log.gz | less -R
  ```

- The Android build system is constantly changing, and the document below is very useful for keeping track of these changes: https://android.googlesource.com/platform/build/+/master/Changes.md
- AOSP Continuous Integration Environment: https://ci.android.com/builds/branches/aosp-master/grid

Emulating Android

- The Android operating system image can be emulated in a virtual machine called Android Virtual Device (AVD).
  - There are two implementations: Android Emulator (Goldfish) and Cuttlefish.
- The Android Emulator (Goldfish) is the older implementation, providing an interface through a desktop application to interact with the device.
  - It can be compiled using the sdk_phone_x86 and sdk_phone_x86_64 products.
- Cuttlefish is the newer implementation and faithful to the AOSP behavior, providing an interface to interact with the device via WebRTC.
  - It can be compiled using the aosp_cf_x86_phone and aosp_cf_x86_64_phone products.

Compiling the Android emulator

- Compiling an Android Emulator image:

  ```
  $ source build/envsetup.sh
  $ lunch sdk_phone_x86-eng
  $ m -j
  ```

- Running the emulator:

  ```
  $ emulator
  ```

- More information about the Android Emulator: https://source.android.com/docs/setup/create/avd

Compiling Cuttlefish

- Building a Cuttlefish image:

  $ source build/envsetup.sh

  $ lunch aosp_cf_x86_phone-eng

  $ m -j

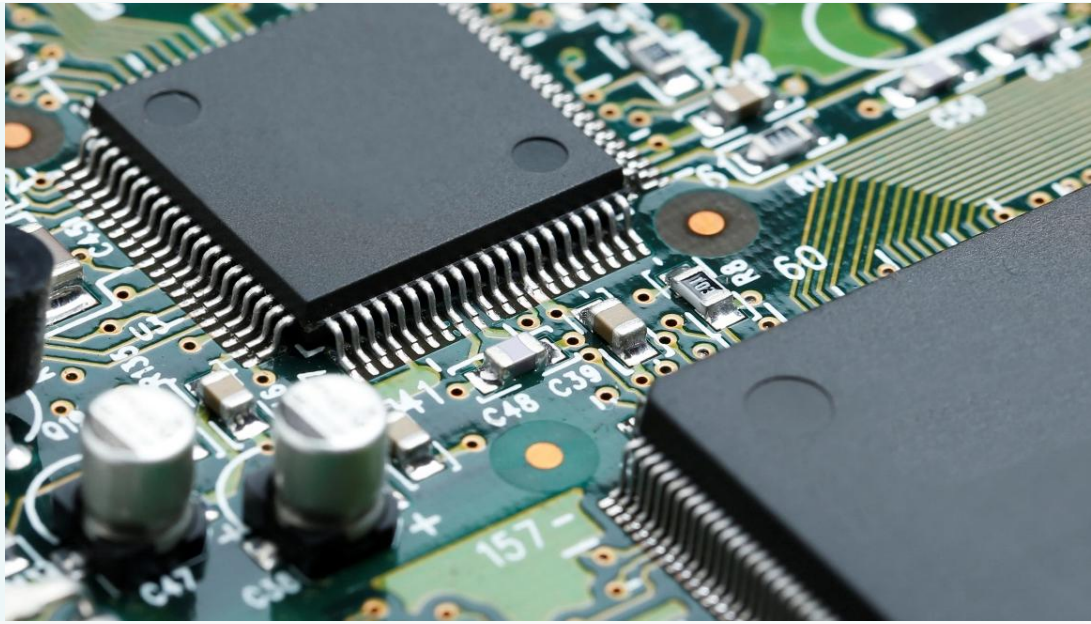
- Running the emulator:

  $ launch_cvd

- More information about Cuttlefish:

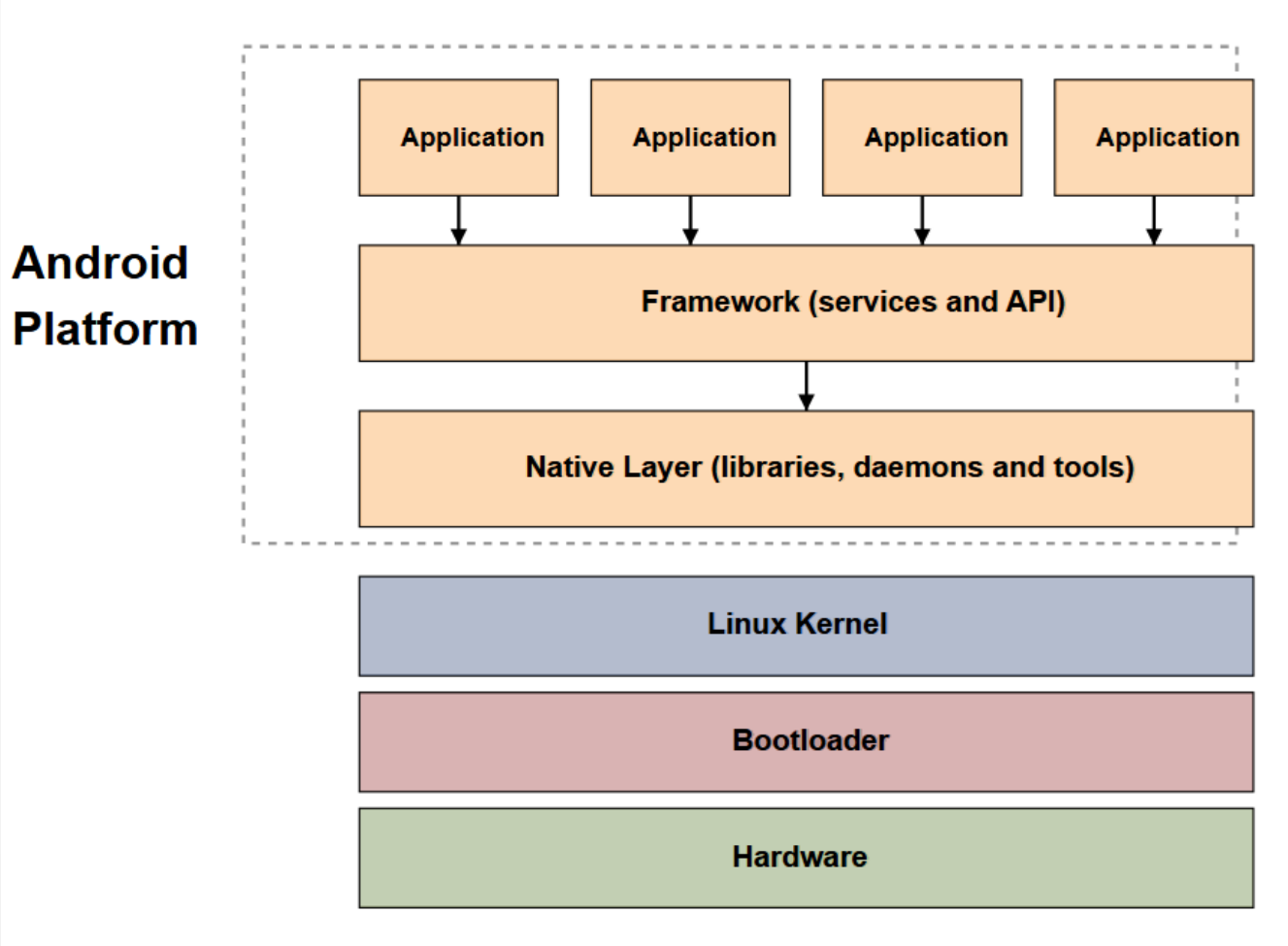
  https://source.android.com/docs/setup/create/cuttlefis

  https://android.googlesource.com/device/google/cuttlefish/

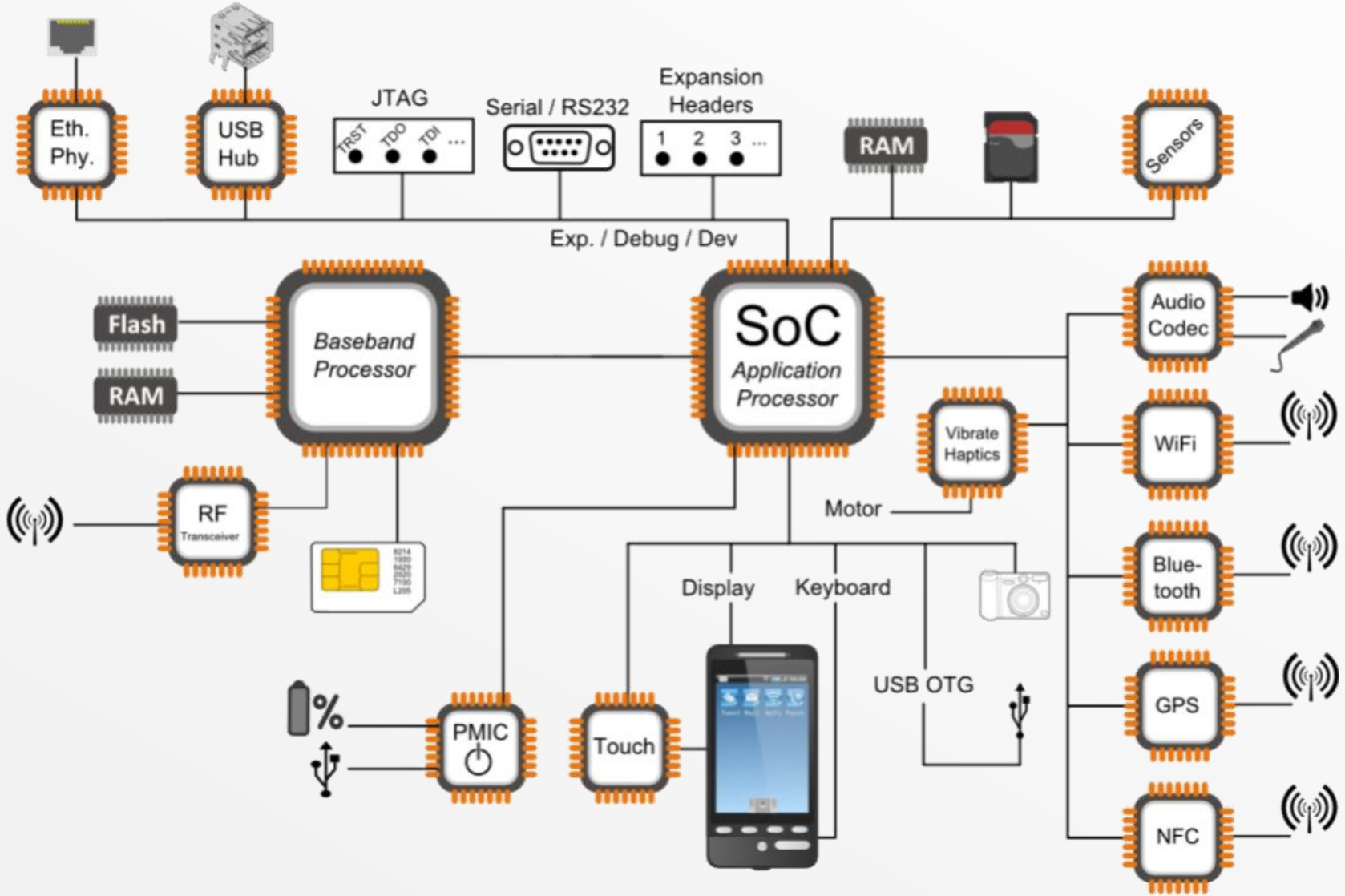


Android build system

Android arhitecture



Typical Android hardware



CPU

- Officially, Android supports ARM and x86 architectures (32-bit and 64-bit).
- In the past, Android has supported other CPU architectures, including MIPS.
- There is an initiative to support Android on RISC-V:
  https://www.youtube.com/watch?v=70O_RmTWP58
  https://ci.android.com/builds/branches/aosp-master/grid
- Currently, it is more common to find Android on 64-bitARM architecture devices (multicore, 1GHz+).

Memory and storage

- Android's RAM and storage requirements can vary greatly,depending on the system being developed.
- Typical RAM requirements:
  - Embedded system: minimum 1G, recommended 2G+.
  - Smartphone/tablet: minimum 2G, recommended 4G+.
- Typical storage requirements (eMMC):
  - Embedded system: 4G+.
  - Smartphone/tablet: 16G+.

Other common features

- Touchscreen display.
- Navigation buttons (Power, Volume, Home, Back, etc.).
  - The buttons can also be emulated in software, or the system can use gesture navigation.
- Sensors (accelerometer, magnetometer, GPS, gyroscope, etc.).
- Wireless communication interfaces (Bluetooth, WiFi, NFC, etc.).
- The CDD defines minimum hardware requirements by device type (handheld, watch, television, automotive, etc.).

https://source.android.com/compatibility/android-cdd

The bootloader

- The bootloader is the code responsible for:
  - Initializing the hardware.
  - Loading and executing the operating system.
- In addition to these basic functions, most bootloaders provide a command terminal for executing different operations, including:
  - Manipulating and configuring the boot process.
  - Formatting the flash drive and writing images.
  - Hardware diagnostic routines.

Bootloader on Android

- No specific bootloader feature is required to boot an Android system.
  - However, the certification process for an Android device may require the existence of some features in the bootloader:
    https://source.android.com/docs/core/architecture/bootloader
- Since the bootloader is entirely hardware-dependent, there is no source code for bootloaders in AOSP.
- Usually, SoC manufacturers provide a reference bootloader for their hardware platforms:

  https://github.com/littlekernel/lk/wiki/Introduction

  https://www.denx.de/wiki/U-Boot

Fastboot

- A feature commonly available in the bootloaders of Android devices is fastboot:
  system/core/fastboot/README.md
- Fastboot is a protocol for communicating with the bootloader via USB or TCP/IP, accessible through a command-line tool called fastboot.
- It has several features, including configuring the device and flashing operating system images.
- Starting with Android 10, some of the fastboot functionality has been moved to the Android user space. https://source.android.com/docs/core/architecture/bootloader/fastbootd

Fastboot tool

```
$ fastboot -h
usage: fastboot [OPTION...] COMMAND...

flashing:
 update ZIP                          Flash all partitions from an update.zip package.
 flashall                            Flash all partitions from $ANDROID_PRODUCT_OUT.
                                     On A/B devices, flashed slot is set as active.
                                     Secondary images may be flashed to inactive slot.
 flash PARTITION [FILENAME]          Flash given partition, using the image from
                                     $ANDROID_PRODUCT_OUT if no filename is given.

basics:
 devices [-l]                        List devices in bootloader (-l: with device paths).
 getvar NAME                         Display given bootloader variable.
 reboot [bootloader]                 Reboot device.

locking/unlocking:
 flashing lock|unlock                Lock/unlock partitions for flashing
 flashing lock_critical|unlock_critical Lock/unlock 'critical' bootloader partitions.
 flashing get_unlock_ability         Check whether unlocking is allowed (1) or not(0).
...
```

The Linux kernel on Android

- Just like most GNU/Linux distributions do, the kernel used in Android is modified to meet the project's needs.
  - These changes are sometimes called Androidisms.
- The changes to the kernel are so significant that the Android user layer only works with these changes!
- Most of the changes are already integrated into the official kernel version, making it possible to run an Android system on a mainline kernel.
- You still need to adapt the configuration to enable some features needed to run Android (Binder, Ashmem, etc).

Binder

- Binder is the main IPC and RPC mechanism in Android, adding the capability of interprocess communication and remote method invocation to the kernel.
- In Android, all communication between applications, services, some daemons and HALs happens via Binder.
- The Binder interface is exposed to the user in files in /dev, accessed via ioctl() calls.

```
# ls /dev/*binder
/dev/binder     /dev/hwbinder     /dev/vndbinder
```

Ashmem

- Ashmem (Anonymous Shared Memory) is a mechanism for sharing memory between processes used in Android.
- Some shortcomings in the standard Linux memory sharing mechanism (POSIX SHM), such as the possibility of resource leaks, led to the creation of this new interface by Google.
  - For example, this implementation uses a reference counter to destroy memory regions that are no longer being used by any process.
- Through the /dev/ashmem file, an application requests a memory region and shares it with other processes via Binder.

Wakelocks

- The wakelock is an Android power management feature.
- The idea is to put the CPU into low-power mode (sleep mode) whenever possible:
  - When an application wants to keep the CPU on, it must hold a wakelock.
  - When no one is holding a wakelock, the kernel puts the CPU into low-power mode.
- The control is done through files exported by the kernel:

```
# ls /sys/power/wake_*
/sys/power/wake_lock      /sys/power/wake_unlock
```

Low memory killer

- When the system runs out of memory, the Linux kernel runs the OOM (Out-of-Memory) killer by default, which tries to kill some processes to free up memory.
- The OOM Killer is unpredictable and can kill an important operating system process, which is why Google developed the Low Memory Killer (LMK).
- The LMK is triggered before the OOM killer, complementing its operation:
  - It takes into account the priority and idleness of the process.
  - It allows you to notify a process before terminating it.
- In recent versions of Android, much of the Low Memory Killer logic has been moved to a user-space daemon called lmkd.

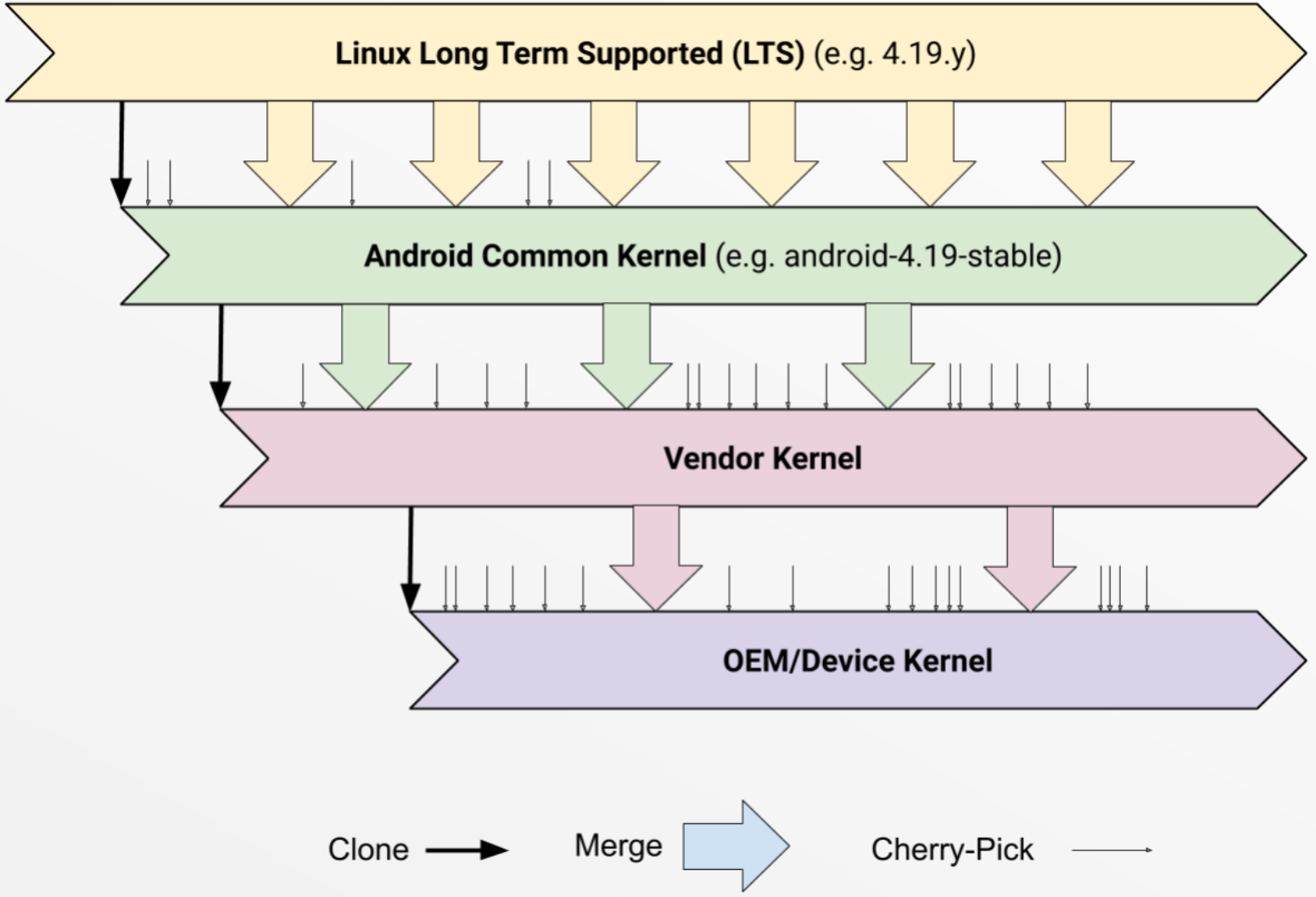
Where does the kernel code come from?

- There is no Linux kernel source code in AOSP!
- But Google provides a base version of the kernel for Android devices in a project called Android Common Kernel (ACK).

  https://source.android.com/devices/architecture/kernel/android-common

  https://android.googlesource.com/kernel/common/
- If the manufacturer of a given hardware platform (vendor) wants to support Android, it will use the ACK as a basis for porting the kernel and providing the BSP to its users (OEM).

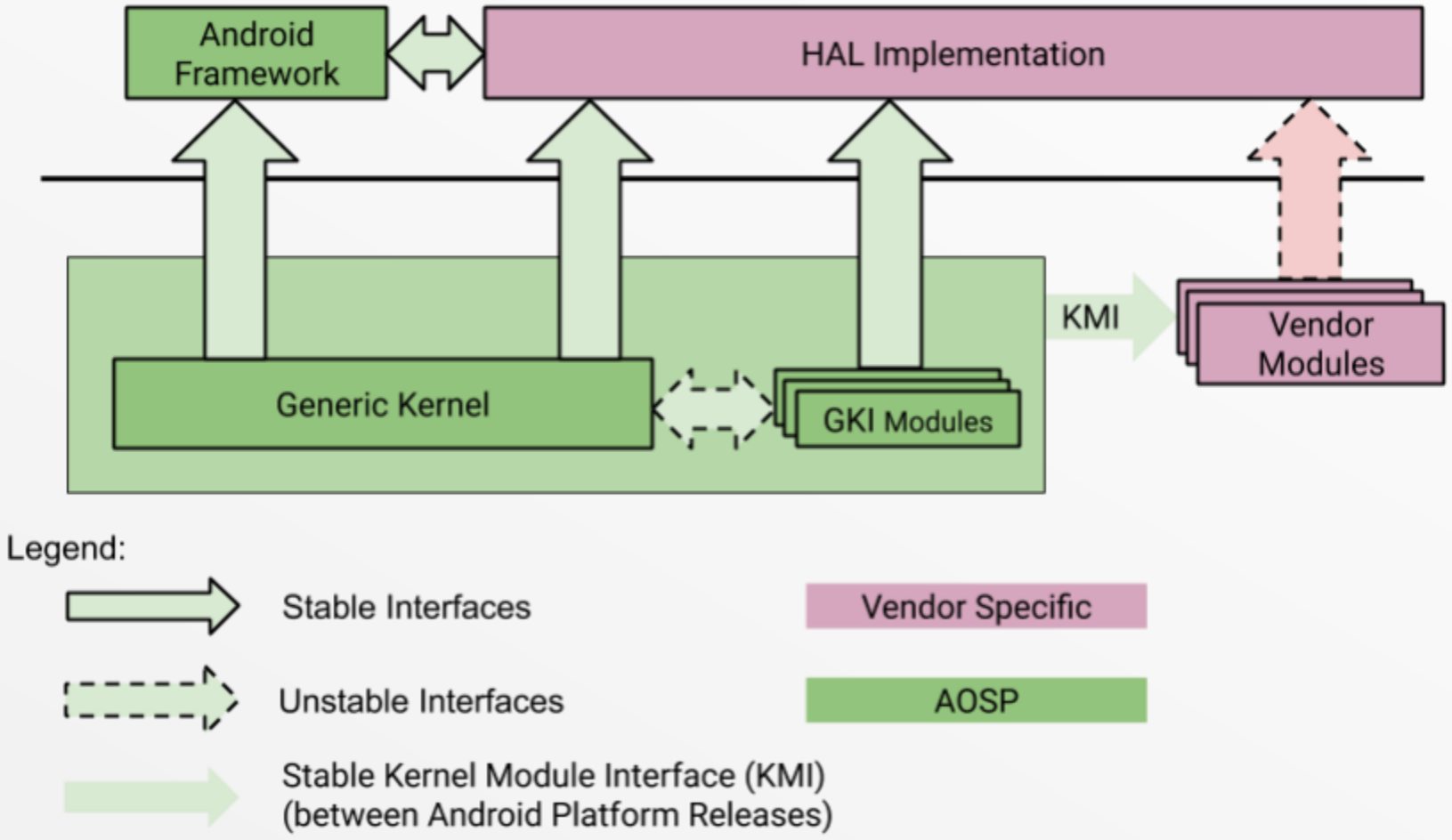
Android kernel forks



Generic kernel image (GKI)

- The excessive number of forks causes a major fragmentation problem, whichmakes it difficult to update the kernel on Android devices in production.
- To solve this problem, Google created the Generic Kernel Image (GKI) project.
  https://source.android.com/devices/architecture/kernel/generic-kernel-image
- The idea is that all hardware manufacturers use the standard kernel provided by Google (ACK), and extend the kernel through loadable modules, using a defined and standardized interface (KMI - Kernel Module Interface).
- In practice, you should only worry about this if you are porting the kernel to run Android on your device, with the aim of certifying it.

GKI architecture



Userspace on Android

- In general, the user space on a Linux system contains the libraries and applications that make up the operating system.
  - On Android, it is also called the Android platform.
- The user space on Android is made up of three main layers:
  - Native layer: native code (C, C++, Rust) that runs outside the virtual machine.
  - Framework: services and APIs that run inside the virtual machine.
  - Applications: Android applications.
- We will study these layers in detail throughout the training.

File organization

- In Linux systems, user layer components are organized according to the FHS (Filesystem Hierarchy Standard) and LSB (Linux Standard Base) standards.

  http://www.pathname.com/fhs/
  https://refspecs.linuxfoundation.org/lsb.shtml
- Most Linux systems comply with these standards:
  - It enables interoperability of applications across different Linux systems.
  - It makes it easier for users to adapt when migrating to other Linux distributions.
- But Android is an exception to the rule!

Organizing files on Android

- Where are important directories like /lib and /usr?

```
# ls /
acct                  debug_ramdisk  odm                    sys
apex                  dev            odm_dlkm               system
bin                   etc            oem                    system_dlkm
bugreports            init           postinstall            system_ext
cache                 init.environ.rc proc                  vendor
config                linkerconfig   product                vendor_dlkm
d                     lost+found     sdcard
data                  metadata       second_stage_resources
data_mirror           mnt            storage
```

- They don't exist!
- To understand how file organization works on Android, it is necessary to study the different partitions that make up its file system.

Partitions

- With the main objective of facilitating maintenance and the update process, the Android file system is very modular and divided into different partitions.
  - Code and data are separated into different partitions.
  - Code is separated into partitions by "source" (Google, vendor, OEM, operator).
  - Code is divided into different partitions to facilitate updates (boot, kernel modules, radio, etc.).
  - A/B update mechanism (seamless system updates) duplicates the partitions.
- Detailed documentation about the main partitions on the project website.https://source.android.com/devices/bootloader/partitions

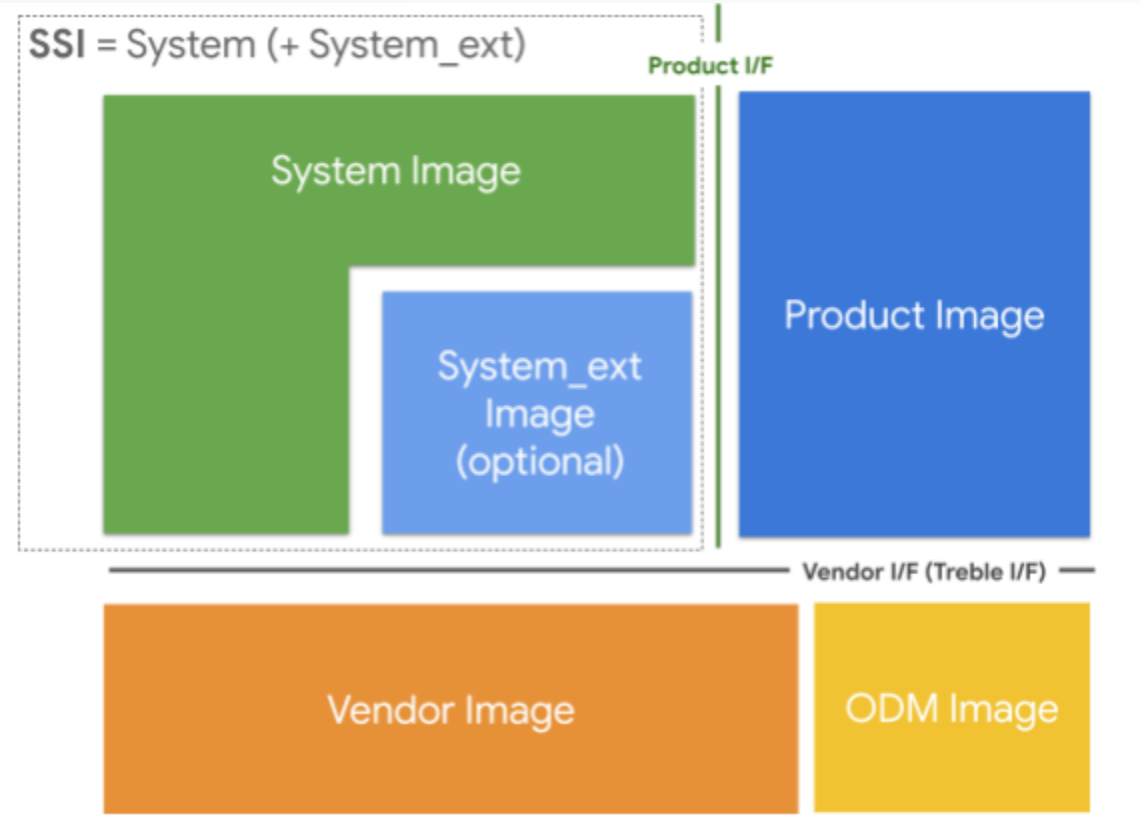
Partitions in Android 13

Partition	Description
boot	kernel image (GKI)
init_boot	generic ramdisk image
vendor_boot	extension of the ramdisk image provided by the manufacturer
system	base operating system image (AOSP)
system_ext	extension of the system partition with components from the manufacturer
vendor	software artifacts from the SoC manufacturer (Qualcomm, Mediatek, etc.)
vendor_dlkm	kernel modules from the SoC manufacturer
odm software	artifacts from the integrator (Motorola, Samsung, etc.)
odm_dlkm	kernel modules from the integrator

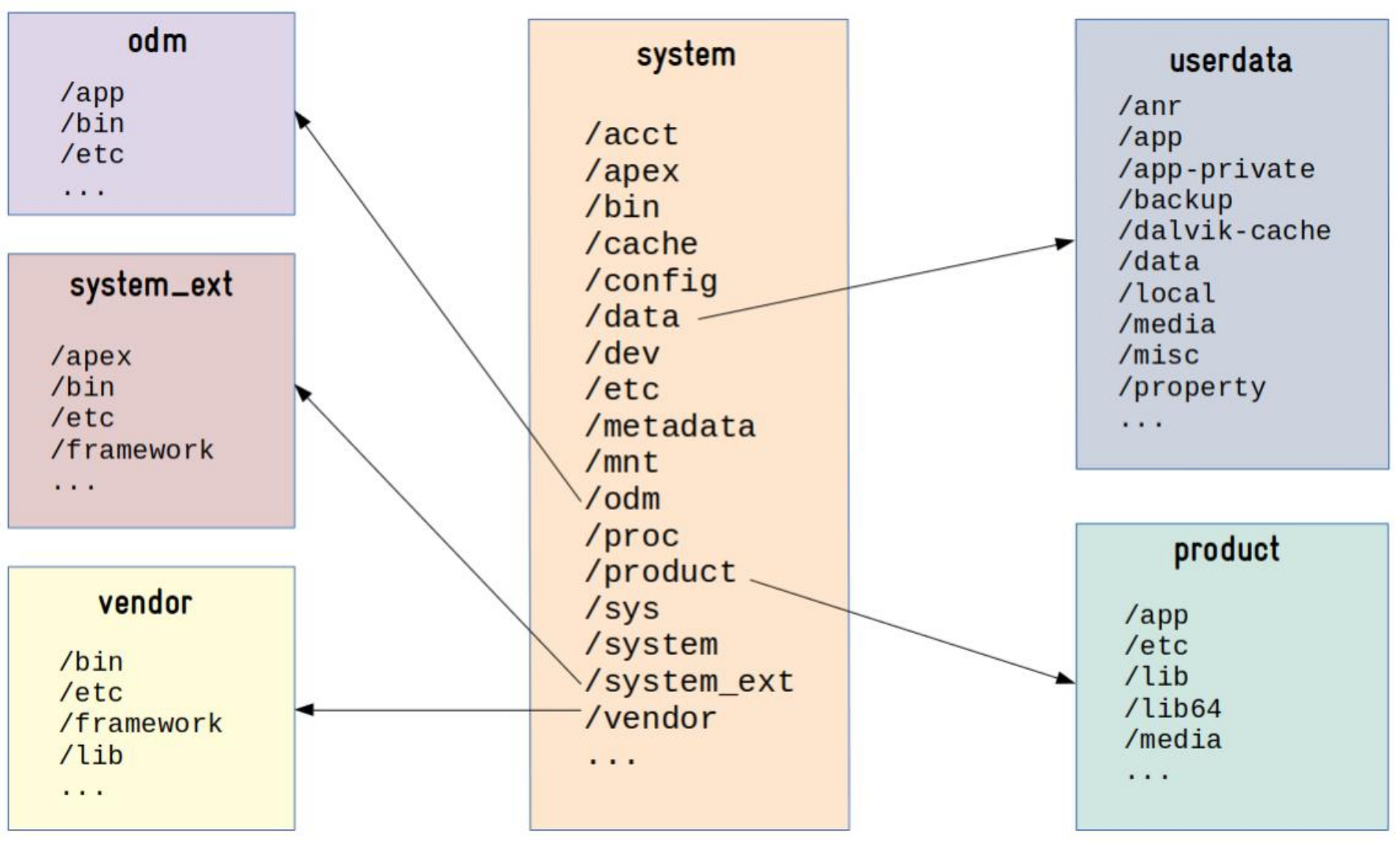
Partitions in Android 13

Partition	Description
product	artifacts of the product's software, used for example to differentiate operator builds
userdata	configuration files, installed applications and other data generated at runtime
metadata	information used in the partition encryption process
recovery	recovery image used in the update process (OTA)
misc	partition used exclusively by the recovery image
cache	temporary data such as downloads or update images
radio	software specific to the radio (if the device has this peripheral)

Main system images



Partitions and mount point



Partition contents

```
# ls system
apex       bin                etc        framework          lib64      product              usr        xbin
app        build.prop         fonts      lib                priv-app   system_ext           vendor

# ls system_ext/
apex       bin                etc        framework          lib64      lost+found           priv-app

# ls /vendor
apex       build.prop         framework  lib64              odm        pci.ids
bin        etc                lib        lost+found         overlay    priv-app

# ls /odm
etc        lib        lost+found

# ls /product
app        bin        etc        lib        lib64      lost+found           media      overlay    priv-app
```

Main directories

Directory	Description
bin and xbin	native binaries
lib[64]	native libraries
etc	native layer configuration files
framework	framework code
usr	framework configuration files
app and priv-app	Android applications (.apk)
apex	software artifacts in APEX format
media	media files (boot animation, audio, etc)
fonts	system fonts (.otf, .ttf, etc)

Organization of the userspace partition

```
# ls /data
adb                 cache               misc_de             ss
anr                 dalvik-cache        nativetest          system
apex                data                nativetest64        system_ce
app                 drm                 nfc                 system_de
app-asec            fonts               ota                 tombstones
app-ephemeral       gsi                 ota_package         unencrypted
app-lib             gsi_persistent_data per_boot            user
app-private         incremental         preloads            user_de
app-staging         local               property            vendor
backup              lost+found          resource-cache      vendor_ce
benchmarktest       media               rollback            vendor_de
benchmarktest64     mediadrm            rollback-history
bootanim            misc                rollback-observer
bootchart           misc_ce             server_configurable_flags
```

Generic system image (GSI)

- GSI (Generic System Image) is a generic image of the system partition that every Android device version >= 8.1 should be able to run.

  https://source.android.com/setup/build/gsi

- The main use of this image is to run certification tests (CTS and VTS).
  - An Android system image is replaced with GSI and tested with CTS and VTS to verify that the interfaces implemented by the manufacturer are compatible with the latest version of Android.
- The GSI can be downloaded from the Android Continuous Integration Environment: https://ci.android.com/builds/branches/aosp-android13-gsi/grid

Android security

- Android's design has a strong focus on operating system security:
  https://source.android.com/docs/security
- Among the operating system's main security features, we have:
  - Only the kernel and a small set of programs run with root access.
  - Use of SELinux to control access to operating system resources.
  - Verification of the integrity and authenticity of images (verified boot).
  - Encryption of the data partition.
  - Integration with a secure execution environment (TEE) and the availability ofa virtualization infrastructure (AVF).

Remote update

- Android has an integrated mechanism for remote updates (OTA updates):

  https://source.android.com/docs/core/ota

- Some of the main features provided by the Android update mechanism:
  - Security in the transport and signature verification of update images.
  - A/B update mechanism (seamless system updates).
  - Dynamic partition creation.
  - Update of individual components (via Google Play) through APK (Android Package) and APEX (Android Pony EXpress) file containers.

ADB

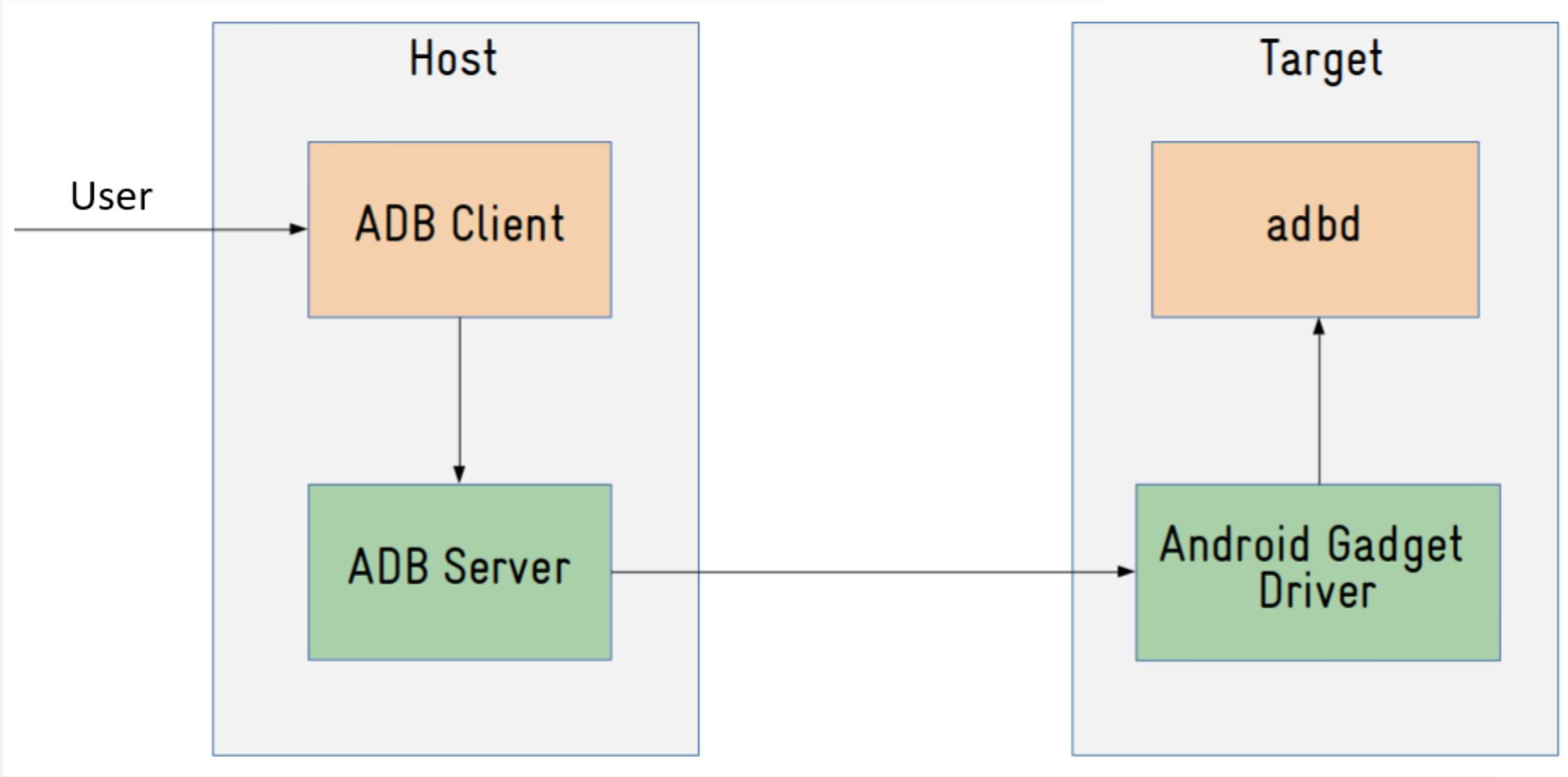
- Usually, the debugging process in embedded systems occurs via a serial port or JTAG interface (for low-level debugging).
- However, mobile and consumer devices such as cell phones and tablets do not usually have these interfaces.
  - The most common is to have only one USB port available.
- This is why Google developed ADB (Android Debug Bridge).

https://developer.android.com/tools/adb

ADB components

- ADB is a debugging tool that can work via USB or TCP/IP.
- It consists of four main components:
  - **ADB client**: runs on the host and serves as an interface (text mode) for executing commands on the target.
  - **ADB server**: runs on the host and manages communication between ADB clients and the ADB daemon on the target.
  - **Android USB Gadget**: only in the case of USB connections, it is a driver in the kernel that receives messages sent by the ADB client and forwards them to the ADB daemon.
  - **ADB daemon**: runs on the target, receives and processes commands received from the ADB client on the host.

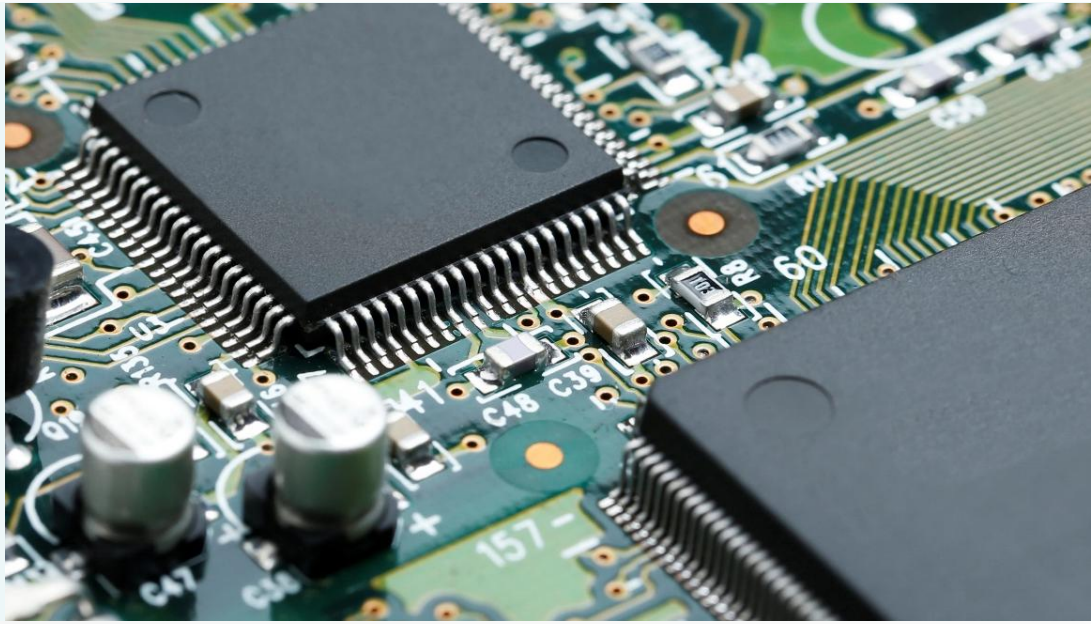
ADB Architecture



Some useful commands

Command	Description
devices	lists available devices
logcat	displays operating system logs
shell	executes a command or starts a terminal session
root	enables root access (for userdebug builds)
push and pull	transfers files between the host and the target
install and uninstall	manages the installation of Android applications(APK)
remount	remounts partitions with write permissions
sync	synchronizes build files with the target
help	displays all available options

Products



What is a product?

- A product defines all the information needed to build an Android image (hardware architecture, supported peripherals, application set, device-specific configurations, etc.).
- By default, AOSP already comes with some products defined, including the latest Google devices (Pixel), some reference boards (DragonBoard, HiKey, etc.) and emulators for various hardware architectures.
- Since Android customizations are done in the product, creating a product is one of the first tasks in porting Android to a hardware platform.

Product configuration

- To date (Android 13), product configuration is done in Makefiles (*.mk), through the definition of several variables that are used to parameterize the Android build process.
  - In the migration to the future Android build system (Bazel), Makefiles will be replaced by Starlark files.

  https://source.android.com/docs/setup/build/bazel/product_config/introduction
- In AOSP, a product's customization files are stored in a directory in the format device/<company>/<product>.
- The goal is to centralize all image definition and customization files in the product directory, facilitating maintenance and the operating system update process.

Creating a product

- To create a product, the first step is to create a directory in the format device/<company>/<product> or vendor/<company>/<product>:

  $ mkdir -p vendor/labworks/superdroid/

- Within this directory, it is necessary to create (at least) three makefiles:
  - <product_name>.mk: product makefile (image software configurations).
  - BoardConfig.mk: board makefile (image hardware configurations).
  - AndroidProducts.mk: makefile to register the product in the build system.
- Other makefiles can be created to modularize the implementation and facilitate product maintenance.

Product makefile

- The product makefile has the name of the product (e.g. superdroid.mk) and contains the image's software configurations.
- In this makefile, several variables can be used to define the characteristics of the software installed in the image (applications, libraries, configuration files, etc.).
- To implement it, it is common to use the inherit-product macro to inherit a makefile that contains the basic configurations of an Android image.
- Reusing makefiles avoids code duplication and facilitates product maintenance and implementation.

Superdroid.mk

```
$(call inherit-product, device/google/cuttlefish/vsoc_x86/phone/aosp_cf.mk)

PRODUCT_NAME         := superdroid
PRODUCT_DEVICE       := superdroid
PRODUCT_MANUFACTURER := Embedded Labworks
PRODUCT_MODEL        := Superdroid phone

PRODUCT_PACKAGES += \
    SuperDroidLauncher

PRODUCT_COPY_FILES += \
    $(LOCAL_PATH)/gpsreset.sh:$(TARGET_COPY_OUT_VENDOR)/etc/gpsreset.sh

PRODUCT_PACKAGE_OVERLAYS += \
    vendor/labworks/superdroid/overlay
```

Product variables

Variable	Description
PRODUCT_NAME	product name
PRODUCT_PACKAGES	add software components to the image(applications, libraries, etc.)
PRODUCT_COPY_FILES	copy files to the image
PRODUCT_PACKAGE_OVERLAYS	overwrite resource files
PRODUCT_VENDOR_PROPERTIES	set custom properties
PRODUCT_CHARACTERISTICS	device characteristics (automotive, tv, etc.)
PRODUCT_MANIFEST_FILES	product manifest file

Adding software to the image

- Variables that can be used to add software to the image:
  - PRODUCT_PACKAGES: adds software to any variant images.
  - PRODUCT_PACKAGES_DEBUG: adds software to debug variant images.
  - PRODUCT_PACKAGES_ENG: adds software to eng variant images.
- Make sure the concatenation sign (+=) is used in the attribution!

```
PRODUCT_PACKAGES +=
        \LiveWallpapers
        \Gallery2
        \SoundRecorder
```

Adding files to the image

- When defining a product, it is common to need to copy some filesdirectly to the final image, such as:
  - Configuration files (init.rc, vold.fstab, ueventd.rc).
  - Executables and binary files (firmware, shell scripts, boot image, etc.).
- These customizations can be made using the PRODUCT_COPY_FILES variable:

```
PRODUCT_COPY_FILES +=
    \vendor/labworks/superdroid/sensor.rc:vendor/etc/init/sensor.rc
    \vendor/labworks/superdroid/vold.fstab:vendor/etc/vold.fstab
    \$(LOCAL_PATH)/gpsreset.sh:odm/etc/gpsreset.sh
```

Resource overlay

- Resource overlay is a mechanism that allows you to overwrite resources defined by an application without having to change the original source code.
- Resources that can be overwritten include media files, messages, strings, configuration files, etc.
- Overlay is only possible on resource files (.png, .xml, etc).
  - Source code overlay is not possible!
- This functionality has nothing to do with the Runtime Resource Overlay (RRO) mechanism.

  https://source.android.com/docs/core/runtime/rros

Resource overlay

- To perform resource overlay, you need to create the overlay directory andadd the resource files following the same directory structure as theAOSP code:

```
$ tree vendor/labworks/superdroid/overlay
/vendor/labworks/superdroid/overlay/
├── frameworks
│   └── base
│       ├── core
│       │   ├── res
│       │   ├── res
│       │   ├── values
│       │   │   └── config.xml
```

- And then declare the overlay directory in the product makefile with thePRODUCT_PACKAGE_OVERLAYS variable:

```
PRODUCT_PACKAGE_OVERLAYS := vendor/labworks/superdroid/overlay
```

Customization of properties

- Android has the concept of properties, which allow you to collect information and configure the behavior of the operating system.
- Some variables are available to define custom properties, including PRODUCT_VENDOR_PROPERTIES and PRODUCT_PRODUCT_PROPERTIES.
- The properties defined in these variables are added to the build.prop file of the corresponding partition and loaded at system boot.
- For example, including properties in the build.prop of the product partition:

```
PRODUCT_PRODUCT_PROPERTIES +=
    \net.dns1=8.8.8.8
    \net.dns2=8.8.4.4
```

Board configuration

- In addition to the product makefile we just studied, it is also necessary to implement the board makefile (BoardConfig.mk).
- This file contains specific hardware information (CPU architecture, bootloader, kernel, hardware components, etc.).
- There is no complete documentation on the variables that can be defined in this file, and in most cases it is necessary to study the source code of the build system to understand how it works.
- To implement this file, the Android documentation recommends using a BoardConfig.mk from another product as a reference.

  https://source.android.com/docs/setup/create/new-device#makefiles

Example Boardconfig.

```
TARGET_BOARD_PLATFORM := vsoc_x86
TARGET_ARCH := x86
TARGET_ARCH_VARIANT := x86
TARGET_CPU_ABI := x86

TARGET_NATIVE_BRIDGE_ARCH := arm
TARGET_NATIVE_BRIDGE_ARCH_VARIANT := armv7-a-neon
TARGET_NATIVE_BRIDGE_CPU_VARIANT := generic
TARGET_NATIVE_BRIDGE_ABI := armeabi-v7a armeabi

BOARD_USES_GENERIC_KERNEL_IMAGE := true
BOARD_KERNEL_CMDLINE += printk.devkmsg=on

BOARD_USES_VENDORIMAGE := true
BOARD_VENDORIMAGE_FILE_SYSTEM_TYPE := $(TARGET_RO_FILE_SYSTEM_TYPE)

BOARD_USES_GENERIC_AUDIO := false
USE_CAMERA_STUB := true

BOARD_HAVE_BLUETOOTH := true
...
```

Registering the product

- To register the product in the Android build system, you need to create the file AndroidProducts.mk with the following variables:
  - PRODUCT_MAKEFILES: location of the product's makefile.
  - COMMON_LUNCH_CHOICES: name of the product to add to the lunch (optional).
- Example of AndroidProducts.mk:

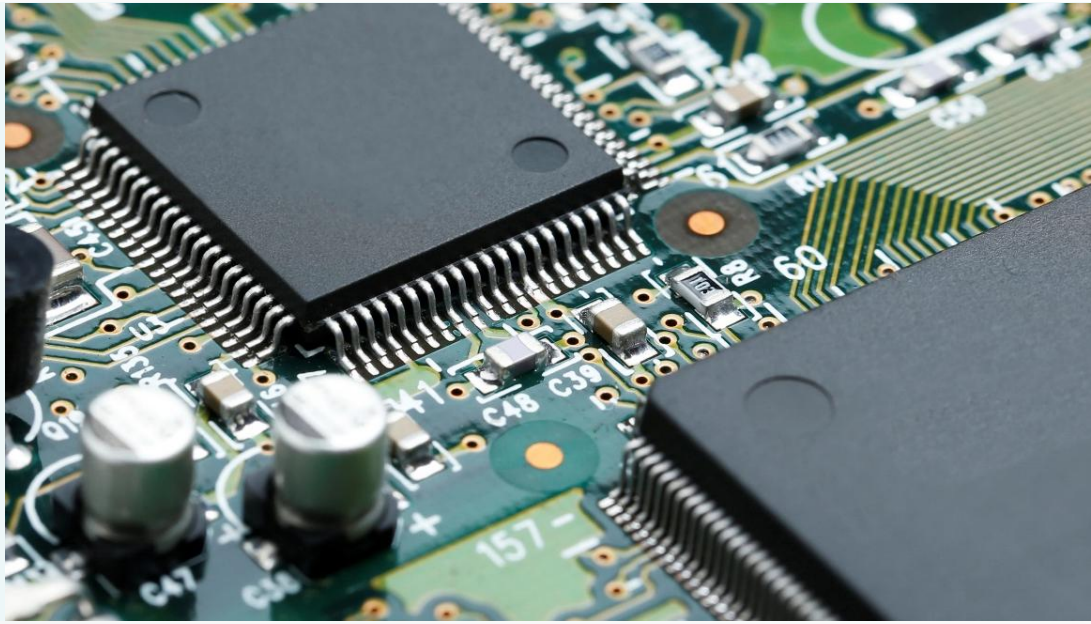
```
PRODUCT_MAKEFILES :=
    \$(LOCAL_DIR)/superdroid.mk
COMMON_LUNCH_CHOICES :=
    \superdroid-eng
    \superdroid-user
```

Summary

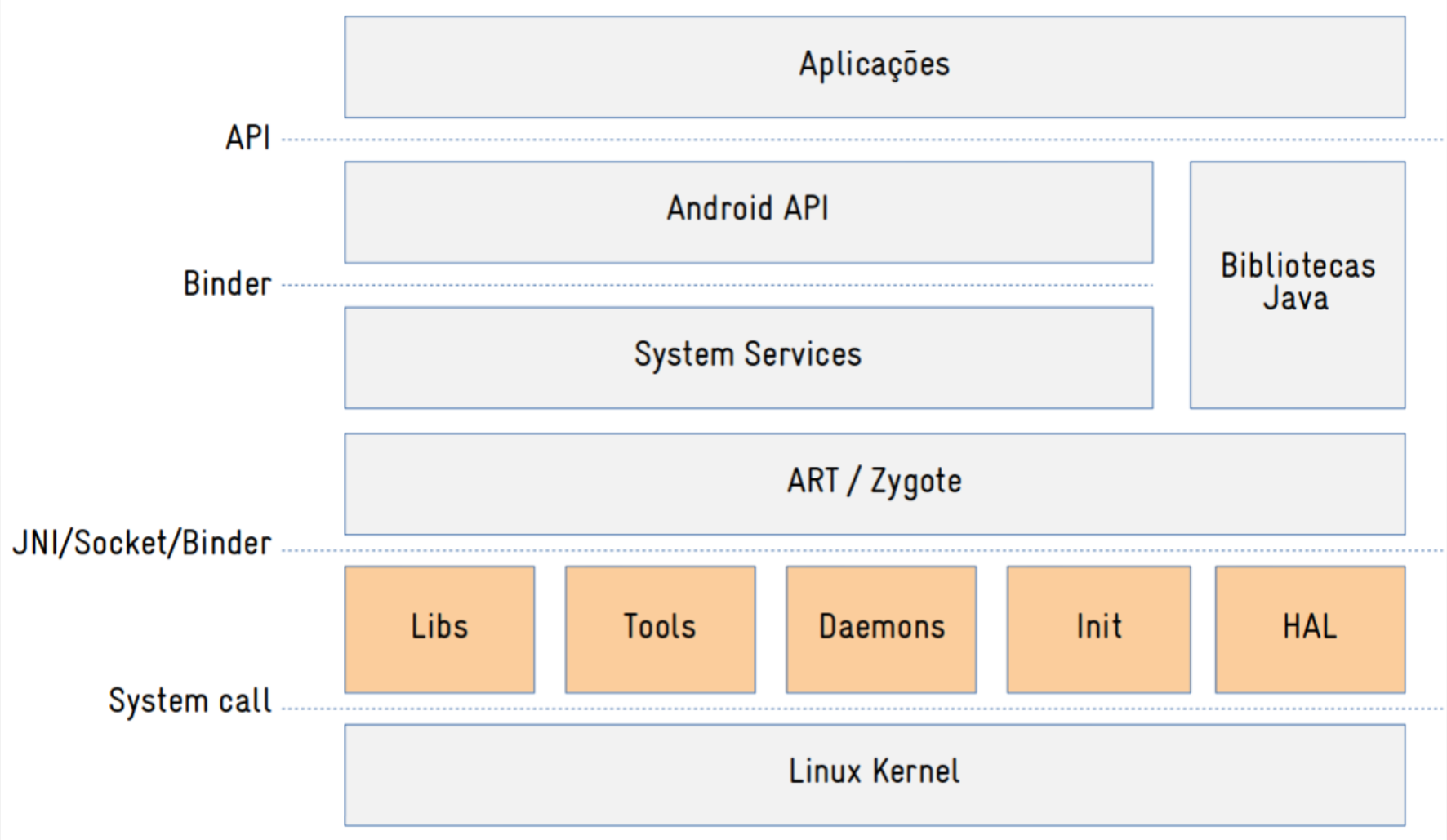
- Create a directory in device/<company>/<product> or vendor/<company>/<product>and deploy the following files:
  - <product_name>.mk: operating system image configuration(software).
  - BoardConfig.mk: board configuration (hardware).
  - AndroidProducts.mk: product registration in the build system.
- Incomplete and outdated documentation is available on the project website:

  https://source.android.com/setup/develop/new-device

Product creation and customization



The Android native layer



The native layer

- The Android native layer is composed of all the user space components that run outside the virtual machine:
  - Libraries (including the system C library - Bionic).
  - Command line tools and utilities.
  - Init mechanism (init process).
  - Hardware abstraction layer (HAL).
  - Daemons.
- We will see in this section of the training that a good part of the operating system's native layer was practically written from scratch for Android!

Libraries

- Android has hundreds of libraries that can be used by applications, divided basically into 3 categories:
  - Popular libraries on Linux systems (boringssl, libjpeg-turbo, sqlite, etc.).
  - Libraries reimplemented with the aim of making the code lighter and simpler to use (tinyalsa, tinyxml2, etc.).
  - Android-specific libraries (libbinder, libutils, liblog, etc.).
- The source code for the libraries is available in the bionic/, system/, external/ and frameworks/native/libs directories.

C library

- One of the main components of a Linux kernel-based system is the C library.
- The C library implements the operating system API, providing an interface for applications to access kernel services through system calls.
- Several C libraries are available for Linux systems, including glibc, uclibc-ng, and musl.
- Android has its own C library, Bionic!

Bionic

- Three factors motivated Google to implement its own C library: license, performance, and size.
- The Bionic implementation is simple, lightweight, and released under the BSD license (source code at bionic/).
- It does not have full POSIX support, which can make it difficult to port native Linux applications to Android.

  platform/bionic/+/refs/tags/android-13.0.0_r44/docs/status.md
- More information about Bionic: https://en.wikipedia.org/wiki/Bionic_(software)

Busybox

- A Linux system requires several basic applications such as an init program, a shell and utilities to manipulate and configure the system.
- Usually, these applications are provided separately by several open source packages (coreutils, bash, grep, sed, tar, wget, modutils, etc.), however:
  - These packages were not developed with a focus on saving resources.
  - It would be quite a lot of work to compile each of these packages separately.
- On Linux systems, BusyBox is a solution to this problem, integrating several tools and applications into a single package, with a focus on saving resources.

Busybox commands

```
addgroup, adduser, adjtimex, ar, arp, arping, ash, awk, basename, bbconfig, bbsh, brctl, bunzip2,
busybox, bzcat, bzip2, cal, cat, catv, chat, chattr, chcon, chgrp, chmod, chown, chpasswd, chpst,
chroot, chrt, chvt, cksum, clear, cmp, comm, cp, cpio, crond, crontab, cryptpw, cttyhack, cut,
date, dc, dd, deallocvt, delgroup, deluser, depmod, devfsd, df, dhcprelay, diff, dirname, dmesg,
dnsd, dos2unix, dpkg, dpkg_deb, du, dumpkmap, dumpleases, e2fsck, echo, ed, egrep, eject, env,
envdir, envuidgid, ether_wake, expand, expr, fakeidentd, false, fbset, fbsplash, fdflush,
fdformat, fdisk, fetchmail, fgrep, find, findfs, fold, free, freeramdisk, fsck, fsck_minix,
ftpget, ftpput, fuser, getenforce, getopt, getsebool, getty, grep, gunzip, gzip, halt, hd, hdparm,
head, hexdump, hostid, hostname, httpd, hush, hwclock, id, ifconfig, ifdown, ifenslave, ifup,
inetd, init, inotifyd, insmod, install, ip, ipaddr, ipcalc, ipcrm, ipcs, iplink, iproute, iprule,
iptunnel, kbd_mode, kill, killall, killall5, klogd, lash, last, length, less, linux32, linux64,
linuxrc, ln, load_policy, loadfont, loadkmap, logger, login, logname, logread, losetup, lpd, lpq,
lpr, ls, lsattr, lsmod, lzmacat, makedevs, man, matchpathcon, md5sum, mdev, mesg, microcom, mkdir,
mke2fs, mkfifo, mkfs_minix, mknod, mkswap, mktemp, modprobe, more, mount, mountpoint, msh, mt, mv,
nameif, netstat, nice, nmeter, nohup, nslookup, od, openvt, parse, passwd, patch, pgrep, pidof,
ping, ping6, pipe_progress, pivot_root, pkill, poweroff, printenv, printf, ps, pscan, pwd,
raidautorun, rdate, rdev, readahead, readlink, readprofile, realpath, reboot, renice, reset,
restorecon, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run_parts, runcon, runlevel, runsv,
runsvdir, rx, script, sed, selinuxenabled, sendmail, seq, sestatus, setarch, setconsole,
setfiles, setfont, setkeycodes, setlogcons, setsebool, setsid, setuidgid, sh, sha1sum, showkey,
slattach, sleep, softlimit, sort, split, start_stop_daemon, stat, strings, stty, su, sulogin, sum,
sv, svlogd, swapoff, swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar, taskset, tcpsvd,
tee, telnet, telnetd, test, tftp, tftpd, time, top, touch, tr, traceroute, true, tty, ttysize,
tune2fs, udhcpc, udhcpd, udpsvd, umount, uname, uncompress, unexpand, unig, unix2dos, unlzma,
```

Toolbox and Toybox

- For licensing reasons, instead of BusyBox, Android uses the Tools Toolbox and Toybox, both released under the BSD license.
- Toolbox is a tool implemented by Google and its source code is available in system/core/toolbox/.
- Toybox is a tool implemented by the community and its source code is available in external/toybox/.
- These tools have some limitations, and for this reason it is common to install BusyBox on an Android system, especially during development.

Toolbox and Toybox commands

```
# toolbox
getprop modprobe setprop start stop toolbox

# toybox
[ acpi base64 basename blkdiscard blkid blockdev cal cat chattr chcon chgrp chmod
chown chroot chrt cksum clear cmp comm cp cpio cut date dd devmem df diff dirname
dmesg dos2unix du echo egrep env expand expr fallocate false fgrep file find flock
fmt free freeramdisk fsfreeze fsync getconf getenforce getfattr getopt grep groups
gunzip gzip head help hostname hwclock i2cdetect i2cdump i2cget i2cset iconv id
ifconfig inotifyd insmod install ionice iorenice iotop kill killall ln load_policy
log logname losetup ls lsattr lsmod lsof lspci lsusb makedevs md5sum microcom mkdir
mkfifo mknod mkswap mktemp modinfo modprobe more mount mountpoint mv nbd-client
nc netcat netstat nice nl nohup nproc nsenter od partprobe paste patch pgrep pidof
ping ping6 pivot_root pkill pmap printenv printf prlimit ps pwd pwdx readelf readlink
realpath renice restorecon rev rfkill rm rmdir rmmod rtcwake runcon sed sendevent
seq setenforce setfattr setsid sha1sum sha224sum sha256sum sha384sum sha512sum
sleep sort split stat strings stty swapoff swapon sync sysctl tac tail tar taskset
tee test time timeout top touch tr traceroute traceroute6 true truncate tty tunctl
uclampset ulimit umount uname uniq unix2dos unlink unshare uptime usleep uudecode
uuencode uuidgen vconfig vi vmstat watch wc which whoami xargs xxd yes zcat
```

The init process

- The init application is executed by the kernel right after mounting the file system, and is responsible for initializing the operating system.
- Linux systems have several implementations of the init process, including sysvinit, systemd and upstart.
- Android has its own initialization system!
- We will study the Android initialization process in detail later in the training.

Shell

- Android uses the MirBSD Korn Shell (mksh) as its terminal program. http://www.mirbsd.org/mksh.htm
- It is much more limited than the shell available on a desktop Linux distribution (e.g.,Bash).
- Documentation can be accessed in the Android source code with the command below:

```
$ man external/mksh/src/mksh.1
```

Daemons

- Daemons are processes that run in the background and are responsible for managing some system functionality:
  - They are executed at startup by the init process.
  - They run in the background for the entire time the system is functional.
  - They are normally used to control and centralize access to a system resource.
- In Android, many daemons serve as an interface between the framework services (System Services) and the resources provided by the operating system (network, storage, energy, etc.), abstracting access to a given resource.

Running daemons

```
# ps -A
```

USER	PID	PPID	VSZ	RSS	WCHAN	ADDR	S NAME
root	1	0	10996412	9020	do_epoll_wait	0	S init
root	178	1	10828020	4568	do_sys_poll	0	S ueventd
logd	238	1	10920556	6132	sigsuspend	0	S logd
lmkd	239	1	10820664	3436	do_epoll_wait	0	S lmkd
root	244	1	10868892	7072	binder_wait_for_work	0	S vold
keystore	268	1	10942252	9996	binder_wait_for_work	0	S keystore2
tombstoned	304	1	10814252	2008	do_epoll_wait	0	S tombstoned
statsd	400	1	10888072	4256	do_epoll_wait	0	S statsd
root	401	1	10996204	10248	binder_wait_for_work	0	S netd
audioserver	481	1	11030404	18648	binder_wait_for_work	0	S audioserver
credstore	482	1	10845068	4904	binder_wait_for_work	0	S credstore
root	502	1	10863736	5832	do_epoll_wait	0	S adbd
incidentd	509	1	10856976	4200	do_epoll_wait	0	S incidentd
root	510	1	10825908	6256	binder_wait_for_work	0	S installd
root	523	1	10775716	5080	binder_wait_for_work	0	S storaged
wifi	524	1	10955548	5504	do_epoll_wait	0	S wificond
radio	529	1	11019012	11316	binder_wait_for_work	0	S libcuttlefish-rild
mdnsr	531	1	10771572	3192	do_select	0	S mdnsd
system	532	1	10906376	4960	binder_wait_for_work	0	S gatekeeperd
root	533	1	11014796	8156	do_epoll_wait	0	S update_engine

Uevent

- Ueventd is responsible for managing the connection of hardware devices on Android (device hotplugging).
  - It is the equivalent of udev or mdev on Linux systems.
- Ueventd features:
  - Loading the firmware of hardware devices.
  - Managing the creation and permission of files in /dev and /sys.
- Parameterizable through configuration

  files:/system/etc/ueventd.rc
  /vendor/etc/ueventd.rc/
  odm/etc/ueventd.rc

/system/etc/ueventd.rc

firmware_directories /etc/firmware/ /odm/firmware/ /vendor/firmware/ /firmware/image/

```
# <path>          <permission>   <user>     <group>
/dev/uhid              0660    uhid       uhid
/dev/uinput            0660    uhid       uhid
/dev/rtc0              0640    system     system
/dev/tty0              0660    root       system
/dev/graphics/*          0660    root       graphics
/dev/input/*           0660    root       input
/dev/v4l-touch*          0660    root       input
/dev/snd/*             0660    system     audio
/dev/bus/usb/*           0660    root       usb
/dev/mtp_usb             0660    root       mtp
/dev/usb_accessory       0660    root       usb
/dev/tun               0660    system     vpn

# <path>                        <file>     <permission>  <user>  <group>
/sys/devices/virtual/input/input*   enable      0660        root    input
/sys/devices/virtual/input/input*   poll_delay  0660        root    input
...
```

Vold

- vold (Volume Daemon) is responsible for managing storage devices on Android.
- Its main features include:
  - Formatting storage devices.
  - Automatically mounting partitions.
  - Managing encryption of partitions, files and directories.
- It can be parameterized through fstab files.

Rild

- The RILD (Radio Interface Layer Daemon) is responsible for managing communication with the modem chip (voice and data).
- It has several functions, including:
  - Making and receiving calls.
  - Sending and receiving SMS, MMS, etc.
  - Establishing data communication (GSM, GPRS, etc.).
- It communicates with the radio hardware through a library implemented by the manufacturer (Vendor RIL).

Netd

- netd (Network Management Service Daemon) is responsible for managing network connections on Android.
- Its main responsibilities include:
  - Managing the configuration of network interfaces.
  - Detecting and establishing new network connections.
  - Resolving DNS requests.
  - Configuring tethering.
  - Setting up a PPP connection.

Installd

- The installd (Install Daemon) is responsible for managing the installation of Android applications (.apk), including:
  - Installing and uninstalling applications.
  - Removing application data and cache.
  - Compiling DEX bytecode into a binary (dex2oat) when the ART virtual machine is enabled.
  - Checking the integrity of APK files.
  - Collecting information about installed applications.

Logd

- The logd (Log Daemon) is responsible for managing Android logs.
- Access to the logs is done through sockets exported in /dev/socket/.

```
# ls /dev/socket/log*
/dev/socket/logd  /dev/socket/logdr  /dev/socket/logdw
```

- To read or write to the logs, it is not necessary to access these sockets directly.
- To do this, applications can use the liblog library.
- In a command terminal, the user can write to the log with the log command andaccess the logs through the logcat tool.

Logcat

```
# logcat
08-06 21:48:32.325  1814  1934 D WifiClientModeManager: entering IdleState
08-06 21:48:32.326  1573  1573 I android.hardware.wifi@1.0-service: Wifi HAL started
08-06 21:48:32.327  1814  1934 I WifiVendorHal: Vendor Hal started successfully
08-06 21:48:32.327  1814  1936 I WifiP2pNative: Registering for interface available listener
08-06 21:48:32.328  1814  1936 D WifiP2pNative: P2P InterfaceAvailableListener true
08-06 21:48:32.343 15642 15642 I wpa_supplicant: Processing hidl events on FD 3
08-06 21:48:32.343 15642 15642 I wpa_supplicant: Successfully initialized wpa_supplicant
08-06 21:48:32.344 15642 15642 I wpa_supplicant: rfkill: Cannot open RFKILL control device
08-06 21:48:32.423 15642 15642 I wpa_supplicant: rfkill: Cannot get wiphy information
08-06 21:48:32.432  1573  1573 I android.hardware.wifi@1.0-service: Configured chip in mode 0
08-06 21:48:32.433  1814  1934 D WificondControl: Setting up interface for client mode
08-06 21:48:32.434  1695  1695 I wificond: create scanner for interface with index: 3
08-06 21:48:32.434  1695  1695 I wificond: subscribe scan result for interface with index: 3
08-06 21:48:32.434  1695  1695 E wificond: No Offload Service available
08-06 21:48:32.434  1695  1695 I wificond: Offload HAL not supported
08-06 21:48:32.435  1814  1934 I chatty  : uid=1000(system) WifiStateMachin identical 8 lines
08-06 21:48:32.436  1693  1738 D CommandListener: Clearing all IP addresses on wlan0
08-06 21:48:32.437  1814  1935 I chatty  : uid=1000(system) WifiScanningSer expire 2 lines
08-06 21:48:32.437  1814  1934 I chatty  : uid=1000(system) WifiStateMachin expire 2 lines
08-06 21:48:32.437  1814  1814 I chatty  : uid=1000 system_server expire 2 lines
08-06 21:48:32.437  1814  1934 I chatty  : uid=1000(system) WifiStateMachin expire 4 lines
08-06 21:48:32.437  1814  1935 I chatty  : uid=1000(system) WifiScanningSer expire 1 line
```

Priority and tags

- Logs are displayed by priority and tag (process identifier):
  08-06 21:48:32.434 1695 1695 I wificond: Offload HAL not supported
- Different priorities are available:
  # logcat –help
  …
  where <tag> is a log component tag (or * for all) and priority is
  :V Verbose (default for <tag>)
  D Debug (default for '*')
  I Info
  W Warn
  E Error
  F Fatal
  S Silent (suppress all output)
  …

Filtering Logs

- Filtering logs by tag:

  ```
  # logcat -s ActivityManager
  ```

- Filtering logs by priority:

  ```
  # logcat -s *:E
  ```

- Filtering logs by tag and priority:

  ```
  # logcat -s ActivityManager:W
  ```

Log buffers

- Logs can also be filtered by buffer:

  ```
  # logcat -b radio
  ```

- The following buffers are available:
  - main: main log, used mainly by applications.
  - system: system logs (native layer and Android framework).
  - events: system logs (framework events).
  - radio: radio layer logs.
  - crash: application crash logs.
  - all: displays all buffers.

Tombstoned

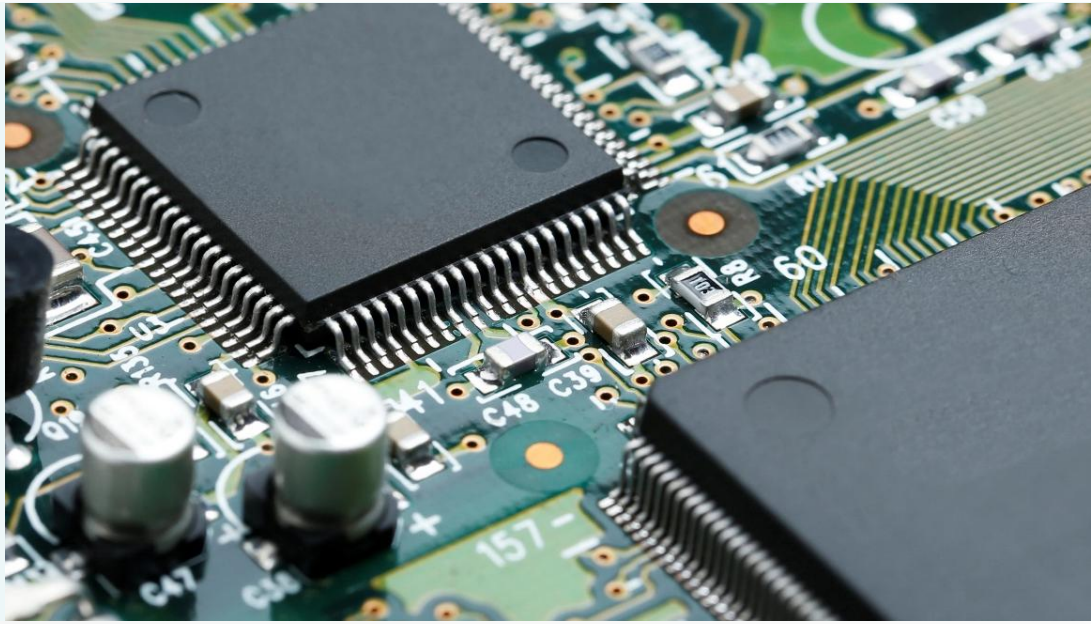
- Tombstoned is an Android debugging daemon.
- It is invoked by the Bionic linker to perform postmortem analysis of a process that terminated unexpectedly.
- It generates tombstone files in /data/tombstones/ for later analysis.

```
# ls /data/tombstones/
tombstone_00 tombstone_01 tombstone_02
```

Other daemons

- storaged: collects and publishes access statistics to storage devices.
  https://source.android.com/devices/tech/debug/storaged
- lmkd: manages memory consumption on Android.
  https://source.android.com/devices/tech/perf/lmkd
- wificond: configures WiFi devices via nl80211 commands.
  https://source.android.com/devices/tech/connect/wifi-overview
- adbd: manages ADB connections.
- incidentd: generates incident (error) reports on framework services.

Native layer commands

- **log**: logs a message to the Android log system.
- **logwrapper**: allows redirecting the output of a command to the Android logs.
- **logcat**: displays system logs.
- **getevent**: monitors events from input devices (keyboard, touchscreen, mouse, button, etc.).
- **sendevent**: generates an input event for the system.
- **debuggerd**: generates a tombstone of any running process
- **lirank**: displays the use of libraries by a running process
- **procrank**: displays a list of processes sorted by memory consumption
- **screencap**: captures the image of the device to a file
- We will have contact with more command line tools from the native layer and the framework during the training.

Q&A