



Android System Development

Android System Development

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Latest update: February 15, 2019.

Document updates and sources:
<https://bootlin.com/doc/training/android>

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com





Rights to copy

© Copyright 2004-2019, Bootlin

License: Creative Commons Attribution - Share Alike 3.0

<https://creativecommons.org/licenses/by-sa/3.0/legalcode>

You are free:

- ▶ to copy, distribute, display, and perform the work
- ▶ to make derivative works
- ▶ to make commercial use of the work

Under the following conditions:

- ▶ **Attribution.** You must give the original author credit.
- ▶ **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
- ▶ For any reuse or distribution, you must make clear to others the license terms of this work.
- ▶ Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

Document sources: <https://github.com/bootlin/training-materials/>



Hyperlinks in the document

There are many hyperlinks in the document

- ▶ Regular hyperlinks:

<https://kernel.org/>

- ▶ Kernel documentation links:

[dev-tools/kasan](#)

- ▶ Links to kernel source files and directories:

[drivers/input/](#)

[include/linux/fb.h](#)

- ▶ Links to the declarations, definitions and instances of kernel symbols (functions, types, data, structures):

[platform_get_irq\(\)](#)

[GFP_KERNEL](#)

[struct file_operations](#)



- ▶ Engineering company created in 2004,
named "Free Electrons" until February 2018.
- ▶ Locations: Orange, Toulouse, Lyon (France)
- ▶ Serving customers all around the world
- ▶ Head count: 13
Only Free Software enthusiasts!
- ▶ Focus: Embedded Linux, Linux kernel, build systems and low level Free and Open Source Software for embedded and real-time systems.
- ▶ Activities: development, training, consulting, technical support.
- ▶ Added value: get the best of the user and development community and the resources it offers.



Bootlin on-line resources

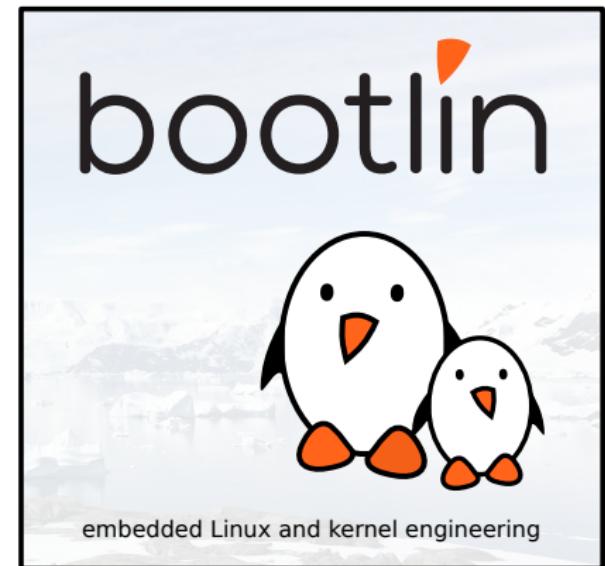
- ▶ All our training materials and technical presentations:
<https://bootlin.com/docs/>
- ▶ Technical blog:
<https://bootlin.com/blog/>
- ▶ News and discussions (LinkedIn):
<https://www.linkedin.com/groups/4501089>
- ▶ Quick news (Twitter):
<https://twitter.com/bootlincom>
- ▶ Elixir - browse Linux kernel sources on-line:
<https://elixir.bootlin.com>



Generic course information

Generic course information

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

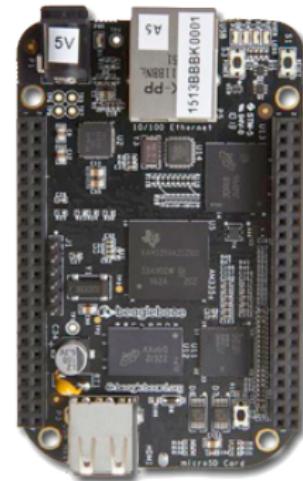




Hardware used in this training session

BeagleBone Black, from CircuitCo

- ▶ Texas Instruments AM335x (ARM Cortex-A8 CPU)
- ▶ SoC with 3D acceleration, additional processors (PRUs) and lots of peripherals.
- ▶ 512 MB of RAM
- ▶ 4 GB of on-board eMMC storage
- ▶ Ethernet, USB host and USB device, microSD, micro HDMI
- ▶ 2 x 46 pins headers, with access to many expansion buses (I2C, SPI, UART and more)
- ▶ A huge number of expansion boards, called *capes*. See
[https://elinux.org/Beagleboard:
BeagleBone_Capes](https://elinux.org/Beagleboard:BeagleBone_Capes).





Course outline - Day 1

Building Android

- ▶ Introduction to Android
- ▶ Getting Android sources
- ▶ Building and booting Android
- ▶ Introduction to the Linux kernel
- ▶ Compiling and booting the Linux kernel

Labs: download Android sources, compile them and boot them with the Android emulator. Recompile the Linux kernel.



Course outline - Day 2

Android kernel, boot and filesystem details

- ▶ Android changes to the Linux kernel
- ▶ Android bootloaders
- ▶ Booting Android
- ▶ Using ADB
- ▶ Android filesystem

Labs: customize, compile and boot Android for the BeagleBone Black board.



Course outline - Day 3

Supporting a new product and customizing it

- ▶ Android build system. Add a new module and product.
- ▶ Android native layer - Bionic, Toolbox, init, various daemons, Dalvik, hardware abstraction, JNI...

Labs: Use ADB, create a new product, customize the product for the BeagleBone Black board.



Course outline - Day 4

Android framework and applications

- ▶ Android framework for applications
- ▶ Introduction to application development
- ▶ Android packages
- ▶ Advise and resources

Labs: compile an external library and a native application to control a USB missile launcher. Create a JNI library and develop an Android application to control the device.



Participate!

During the lectures...

- ▶ Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- ▶ This helps the trainer to detect any explanation that wasn't clear or detailed enough.
- ▶ Don't hesitate to share your experience, for example to compare Linux with other operating systems used in your company.
- ▶ Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- ▶ Your participation can make our session more interactive and make the topics easier to learn.



Practical lab guidelines

During practical labs...

- ▶ We cannot support more than 8 workstations at once (each with its board and equipment). Having more would make the whole class progress slower, compromising the coverage of the whole training agenda (exception for public sessions: up to 10 people).
- ▶ So, if you are more than 8 participants, please form up to 8 working groups.
- ▶ Open the electronic copy of your lecture materials, and use it throughout the practical labs to find the slides you need again.
- ▶ Don't hesitate to copy and paste commands from the PDF slides and labs.



Advise: write down your commands!

During practical labs, write down all your commands in a text file.

- ▶ You can save a lot of time re-using commands in later labs.
- ▶ This helps to replay your work if you make significant mistakes.
- ▶ You build a reference to remember commands in the long run.
- ▶ That's particularly useful to keep kernel command line settings that you used earlier.
- ▶ Also useful to get help from the instructor, showing the commands that you run.

gedit ~/lab-history.txt

Lab commands

```
Cross-compiling kernel:  
export ARCH=arm  
export CROSS_COMPILE=arm-linux-  
make sama5_defconfig  
  
Booting kernel through tftp:  
setenv bootargs console=ttyS0 root=/dev/nfs  
setenv bootcmd tftp 0x21000000 zImage; tftp  
0x22000000 dtb; bootz 0x21000000 - 0x2200...
```

```
Making ubifs images:  
mkfs.ubifs -d rootfs -o root.ubifs -e 124KiB  
-m 2048 -c 1024
```

```
Encountered issues:  
Restart NFS server after editing /etc/exports!
```



Cooperate!

As in the Free Software and Open Source community, cooperation during practical labs is valuable in this training session:

- ▶ If you complete your labs before other people, don't hesitate to help other people and investigate the issues they face. The faster we progress as a group, the more time we have to explore extra topics.
- ▶ Explain what you understood to other participants when needed. It also helps to consolidate your knowledge.
- ▶ Don't hesitate to report potential bugs to your instructor.
- ▶ Don't hesitate to look for solutions on the Internet as well.



Command memento sheet

- ▶ This memento sheet gives command examples for the most typical needs (looking for files, extracting a tar archive...)
- ▶ It saves us 1 day of UNIX / Linux command line training.
- ▶ Our best tip: in the command line shell, always hit the Tab key to complete command names and file paths. This avoids 95% of typing mistakes.
- ▶ Get an electronic copy on https://bootlin.com/doc/legacy/command-line/command_memento.pdf





vi basic commands

- ▶ The vi editor is very useful to make quick changes to files in an embedded target.
- ▶ Though not very user friendly at first, vi is very powerful and its main 15 commands are easy to learn and are sufficient for 99% of everyone's needs!
- ▶ Get an electronic copy on https://bootlin.com/doc/legacy/command-line/vi_memento.pdf
- ▶ You can also take the quick tutorial by running vimtutor. This is a worthy investment!

vi basic commands

Summary of most useful commands

Basic command mode

Repeating commands

Leaving for writing

Moving the cursor

Entering editing mode

Replacing characters, lines and words

Copying and pasting

Deleting lines

Using the command line

Applying a command several times - Examples

Other

Exiting and saving

Using further

Many online resources

Free Resources





Practical lab - Training Setup



Prepare your lab environment

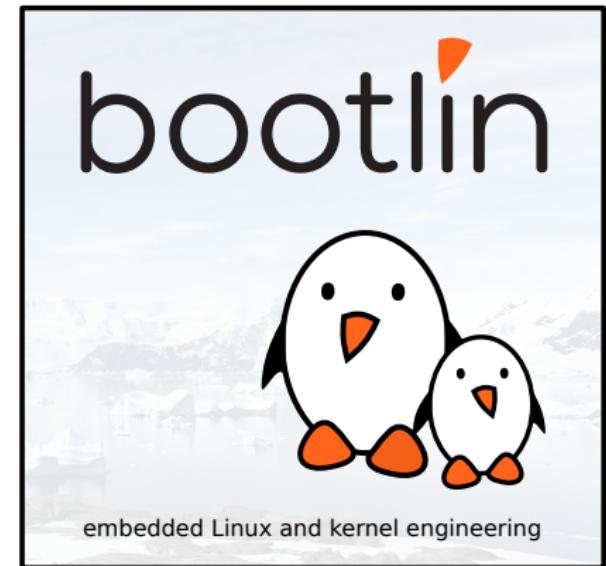
- ▶ Download and extract the lab archive



Introduction to Android

Introduction to Android

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Introduction to Android

Features



Features

- ▶ All you can expect from a modern mobile OS:
 - ▶ Application ecosystem, allowing to easily add and remove applications and publish new features across the entire system
 - ▶ Support for all the web technologies, with a browser built on top of the well-established Blink rendering engine
 - ▶ Support for hardware accelerated graphics through OpenGL ES
 - ▶ Support for all the common wireless mechanisms: GSM, CDMA, UMTS, LTE, Bluetooth, WiFi, NFC.



Introduction to Android

History



Early Years

- ▶ Began as a start-up in Palo Alto, CA, USA in 2003
- ▶ Focused from the start on software for mobile devices
- ▶ Very secretive at the time, even though founders achieved a lot in the targeted area before founding it
- ▶ Finally bought by Google in 2005
- ▶ Andy Rubin, founder of Android, Inc was also CEO of Danger, Inc, a company producing one of the early smartphones, the Sidekick





Opening Up

- ▶ Google announced the Open Handset Alliance in 2007, a consortium of major actors in the mobile area built around Android
 - ▶ Hardware vendors: Intel, Texas Instruments, Qualcomm, Nvidia, etc.
 - ▶ Software companies: Google, eBay, etc.
 - ▶ Hardware manufacturers: Motorola, HTC, Sony Ericsson, Samsung, etc.
 - ▶ Mobile operators: T-Mobile, Telefonica, Vodafone, etc.



Android Open Source Project (AOSP)

- ▶ At every new version, Google releases its source code through this project so that community and vendors can work with it.
 - ▶ One major exception: Honeycomb has not been released because Google stated that its source code was not clean enough to release it.
- ▶ One can fetch the source code and contribute to it, even though the development process is very locked by Google
- ▶ Only a few devices are supported through AOSP though, only the two most recent Android development phones and tablets (part of the Nexus brand) and the pandaboard

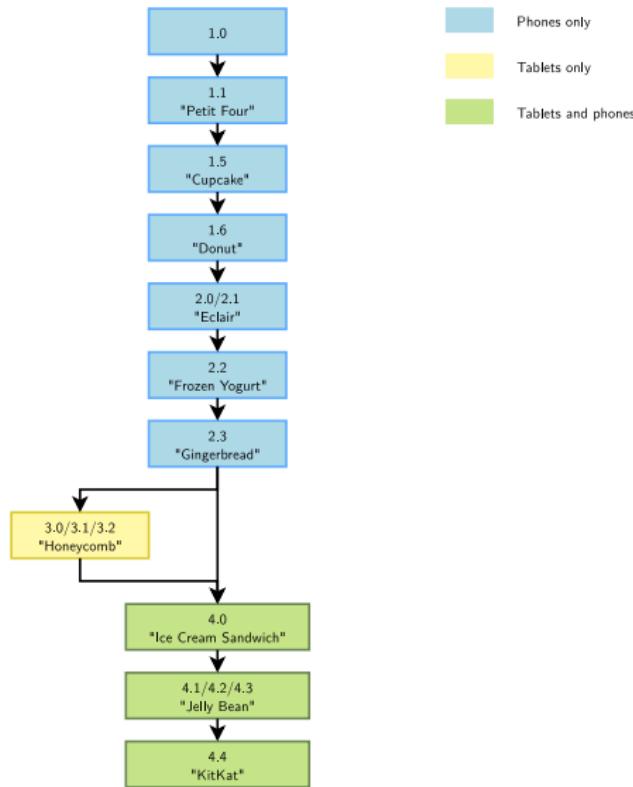


Android Releases

- ▶ Each new version is given a dessert name
- ▶ Released in alphabetical order
- ▶ Latest releases:
 - ▶ Android 2.3 Gingerbread
 - ▶ Android 3.X Honeycomb
 - ▶ Android 4.0 Ice Cream Sandwich
 - ▶ Android 4.1/4.2/4.3 Jelly Bean
 - ▶ Android 4.4 KitKat



Android Versions



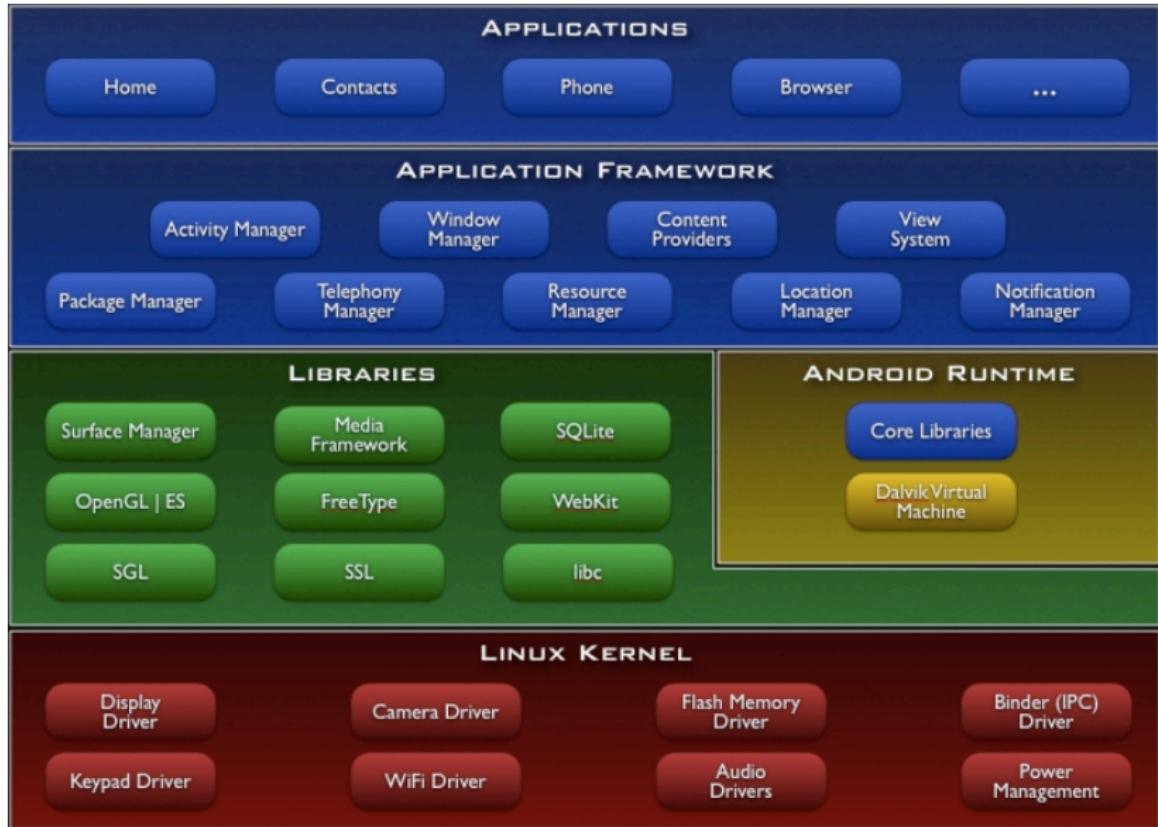


Introduction to Android

Architecture



Architecture





The Linux Kernel

- ▶ Used as the foundation of the Android system
- ▶ Numerous additions from the stock Linux, including new IPC (Inter-Process Communication) mechanisms, alternative power management mechanism, new drivers and various additions across the kernel
- ▶ These changes are beginning to go into the staging/ area of the kernel, as of 3.3, after being a complete fork for a long time



Android Libraries

- ▶ Gather a lot of Android-specific libraries to interact at a low-level with the system, but third-parties libraries as well
- ▶ Bionic is the C library, SurfaceManager is used for drawing surfaces on the screen, etc.
- ▶ But also Blink, SQLite, OpenSSL coming from the free software world



Android Runtime

Handles the execution of Android applications

- ▶ Almost entirely written from scratch by Google
- ▶ Contains Dalvik, the virtual machine that executes every application that you run on Android, and the core library for the Java runtime, coming from Apache Harmony project
- ▶ Also contains system daemons, init executable, basic binaries, etc.



Android Framework

- ▶ Provides an API for developers to create applications
- ▶ Exposes all the needed subsystems by providing an abstraction
- ▶ Allows to easily use databases, create services, expose data to other applications, receive system events, etc.



Android Applications

- ▶ AOSP also comes with a set of applications such as the phone application, a browser, a contact management application, an email client, etc.
- ▶ However, the Google apps and the Android Market app aren't free software, so they are not available in AOSP. To obtain them, you must contact Google and pass a compatibility test.



Hardware Requirements for Android



Android Hardware Requirements

- ▶ Google produces a document updated every new Android version called the Compatibility Definition Document (*CDD*).
- ▶ This document provides all the information you need on the expectations Google have about what should be an Android device
- ▶ It details both the hardware and the global behaviour of the system.
- ▶ While nothing forces you to follow that document if you don't care about the Google applications, it usually gives a good idea of the current hardware requirements.
- ▶ We'll be detailing the requirements for KitKat
- ▶ <http://source.android.com/compatibility/android-cdd.pdf>



SoC requirements

- ▶ Since Android in itself is quite huge, the hardware required is quite powerful.
- ▶ Unlike Linux, Android officially supports only a few architectures
 - ▶ ARM v7a (basically, all the SoCs based on the Cortex-A CPUs)
 - ▶ x86
 - ▶ MIPS
- ▶ You also need to have a powerful enough GPU with OpenGL ES support. Latest versions of Android require the 3D hardware acceleration



Storage and RAM needed

- ▶ The required RAM size is also quite huge, 340MB are required for the kernel and user space memory
- ▶ Required storage is quite huge as well. An image of the system is around 200-300MB, and you must have 350MB of data space for the user plus 1GB of shared storage for the applications.
- ▶ This is the minimum, and Google actually strongly suggest to have at least 2GB dedicated to the applications in order to be able to upgrade to a later version
- ▶ Google recommends to use block devices for storage and not flash devices.
- ▶ The shared space has to be accessible from a host computer by some way, like NFS, USB Mass Storage, MTP, etc.



External Peripherals 1/2

- ▶ No form of communication supported is mandatory, but you need at least one form of data networking with a throughput of at least 200 kbit per second.
- ▶ You will also need obviously a rather large screen with a pointer device, presumably a touchscreen.
- ▶ Screens supported must have a screen size of at least 2.5 inches, with a minimal resolution of 426x320, with a ratio between 4:3 and 16:9 and with a color depth of at least 16bits.



External Peripherals 2/2

- ▶ Sensors are not mandatory, but depending of the class of sensors, they are:
 - ▶ Recommended
 - ▶ Accelerometer
 - ▶ Magnetometer
 - ▶ GPS
 - ▶ Gyroscope
 - ▶ Optional
 - ▶ Barometer
 - ▶ Photometer
 - ▶ Proximity Sensor
 - ▶ Optional but discouraged
 - ▶ Thermometer



Unusual Android Devices: Nook E-Book Reader





Unusual Android Devices: Portable Console





Unusual Android Devices: Microwave Oven





Unusual Android Devices: Treadmill





When to choose Android

- ▶ All of the requirements listed above are only if you want to be eligible to the Android Play Store
- ▶ If you don't want to get the store, you can obviously ignore these
- ▶ However, Android really makes sense in a system that has at least:
 - ▶ A large screen
 - ▶ A powerful SoC, with several CPUs, plenty of RAM and storage space (around 2GB) and a decent GPU
- ▶ This is not an advisable choice when you want to build a headless system, or a cheap system with limited resources
- ▶ In this case, a regular Linux system is definitely more appropriate. It will save you engineering costs, reduce the price of your hardware, and bring the same set of features you could expect from a headless Android



Practical lab - Android Source Code

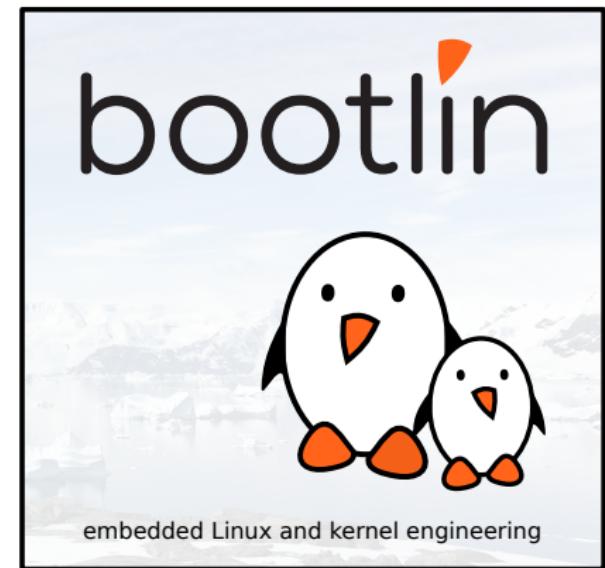


- ▶ Install all the development packages needed to fetch and compile Android
- ▶ Download the `repo` utility
- ▶ Use `repo` to download the source code for Android and for all its components



Android Source Code and Compilation

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





How to get the source code



Source Code Location

- ▶ The AOSP project is available at
<http://source.android.com>
- ▶ On this site, along with the code, you will find some resources such as technical details, how to setup a machine to build Android, etc.
- ▶ The source code is split into several Git repositories for version control. But as there is a lot of source code, a single Git repository would have been really slow
- ▶ Google split the source code into a one Git repository per component
- ▶ You can easily browse these git repositories using
<https://code.google.com/p/android-source-browsing/source/browse/>



Source code licenses

- ▶ Mostly two kind of licenses:
 - ▶ GPL/LGPL Code: Linux
 - ▶ Apache/BSD: All the rest
 - ▶ In the `external` folder, it depends on the component
- ▶ While you might expect Google's apps for Android, like the Android Market (now called Google Play Store), to be in the AOSP as well, these are actually proprietary and you need to be approved by Google to get them.



- ▶ This makes hundreds of Git repositories
- ▶ To avoid making it too painful, Google also created a tool: `repo`
- ▶ Repo aggregates these Git repositories into a single folder from a manifest file describing how to find these and how to put them together
- ▶ Also aggregates some common Git commands such as `diff` or `status` that are run across all the Git repositories
- ▶ You can also execute a shell command in each repository managed by Repo using the `repo forall` command



Repo's manifest

- ▶ repo relies on a git repository that will contain XML files called manifests
- ▶ These manifests gives the information about where to download some source code and where to store it. It can also provide some additional and optional information such as a revision to use, an alternative server to download from, etc.
- ▶ The main manifests are stored in this git repo, and are shared between all the users, but you can add some local manifests.
- ▶ repo will also use any XML file that is under .repo/local_manifests



Manifests syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
    <remote name="github"
        fetch="https://github.com/" />

    <default remote="github" />

    <project name="foo/bar" path="device/foo/bar" revision="v14.42" />

    <remove-project name="foo/bar" />
</manifest>
```



Source code organization



Source Code organization 1/3

- ▶ Once the source code is downloaded, you will find several folders in it

`bionic/` is where Android's standard C library is stored

`bootable/` contains code samples regarding the boot of an Android device. In this folder, you will find the protocol used by all Android bootloaders and a recovery image

`build/` holds the core components of the build system

`cts/` The Compatibility Test Suite

`dalvik/` contains the source code of the Dalvik virtual machine



Source Code Organization 2/3

`development/` holds the development tools, debug applications, API samples, etc

`device/` contains the device-specific components

`docs/` contains HTML documentation hosted at
`http://source.android.com`

`external/` is one of the largest folders in the source code, it contains all the external projects used in the Android code

`frameworks/` holds the source code of the various parts of the framework

`hardware/` contains all the hardware abstraction layers



Source Code Organization 3/3

`libcore/` is the Java core library

`libnativehelper/` contains a few JNI helpers for the Android base classes

`ndk/` is the place where you will find the Native Development Kit, which allows to build native applications for Android

`packages/` contains the standard Android applications

`prebuilt/` holds all the prebuilt binaries, most notably the toolchains

`sdk/` is where you will find the Software Development Kit

`system/` contains all the basic pieces of the Android system: init, shell, the volume manager, etc.

- ▶ You can get a more precise description at
<http://elinux.org/Master-android>



Compilation



Android Compilation Process

- ▶ Android's build system relies on the well-tried GNU/Make software
- ▶ Android is using a "product" notion which corresponds to the specifications of a shipping product, i.e. *crespo* for the Google Nexus S vs *crespo4g* for the Sprint's Nexus S with LTE support
- ▶ To start using the build system, you need to include the file `build/envsetup.sh` that defines some useful macros for Android development or sets the `PATH` variable to include the Android-specific commands
- ▶ `source build/envsetup.sh`



Prepare the process

- ▶ Now, we can get a list of all the products available and select them with the `lunch` command
- ▶ `lunch` will also ask for a build variant, to choose between `eng`, `user` and `userdebug`, which corresponds to which kind of build we want, and which packages it will add
- ▶ You can also select variants by passing directly the combo `product-variant` as argument to `lunch`



Compilation

- ▶ You can now start the compilation just by running `make`
- ▶ This will run a full build for the currently selected product
- ▶ There are many other build commands:

`make <package>` Builds only the package, instead of going through the entire build

`make clean` Cleans all the files generated by previous compilations

`make clean-<package>` Removes all the files generated by the compilation of the given package

`mm` Builds all the modules in the current directory

`mmm <directory>` builds all the modules in the given directory



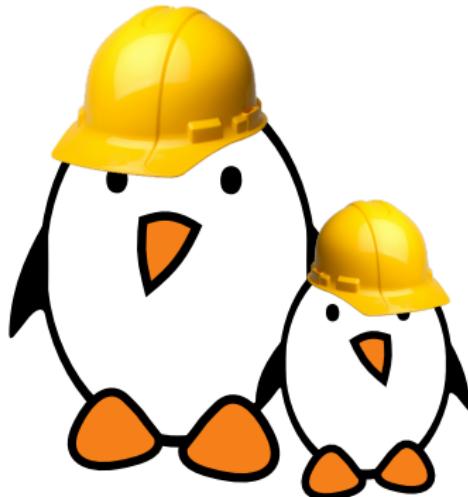
Contribute



- ▶ For the Android development process, Google also developed a tool to manage projects and ease code reviews.
- ▶ It once again uses Git to do so and Repo is also built around it so that you can easily contribute to Android
- ▶ To do so, start a new branch with `repo start <branchname>`
- ▶ Do your usual commits with Git
- ▶ When you are done, upload to Gerrit using `repo upload`



Practical lab - First Compilation



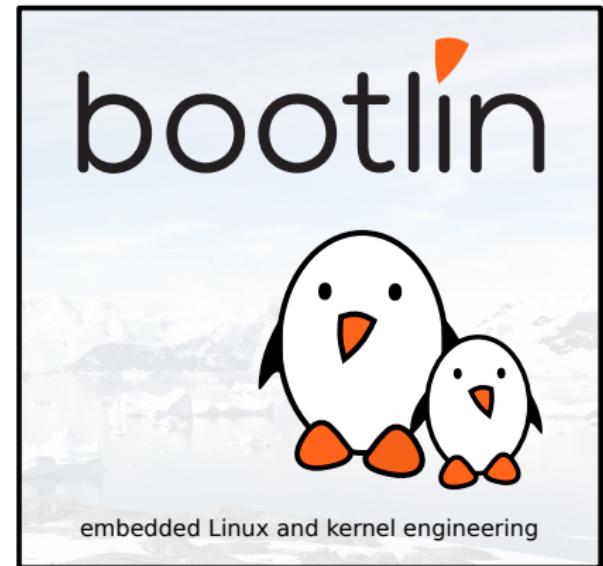
- ▶ Configure which system to build Android for
- ▶ Compile your first Android root filesystem



Linux kernel introduction

Linux kernel introduction

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Linux features



History

- ▶ The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.
- ▶ The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
 - ▶ Linux quickly started to be used as the kernel for free software operating systems
- ▶ Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- ▶ Nowadays, more than one thousand people contribute to each kernel release, individuals or companies big and small.

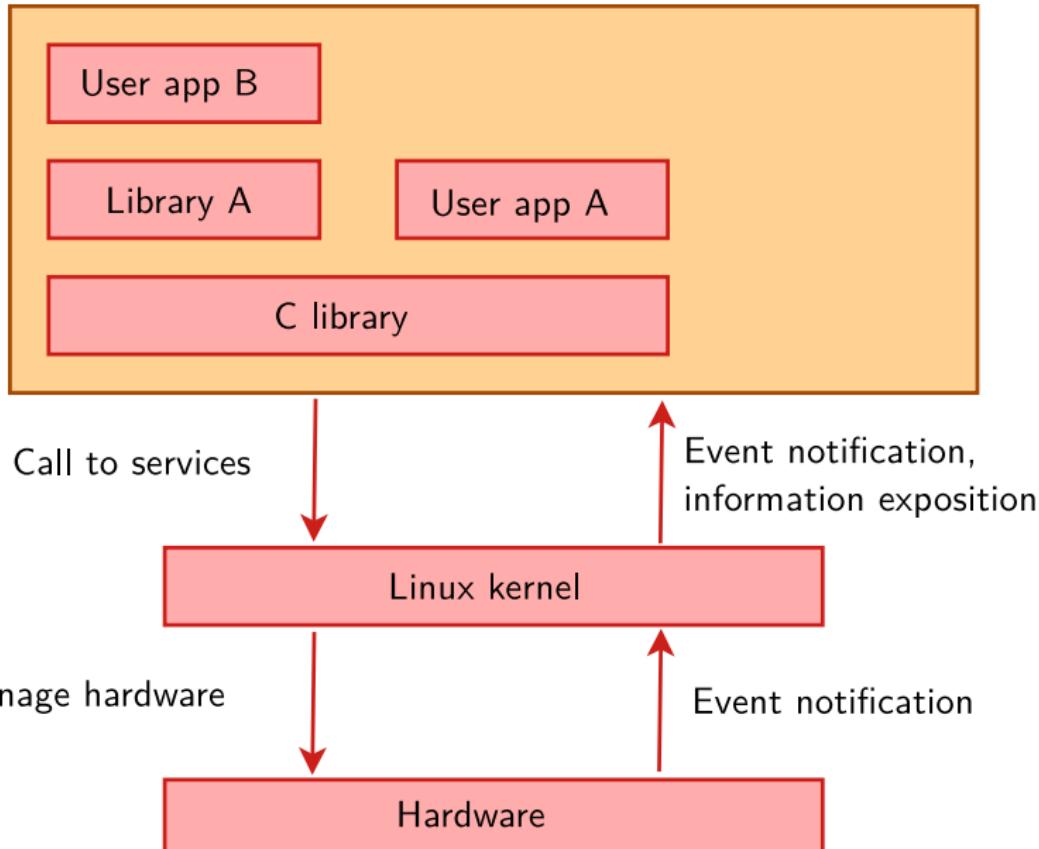


Linux kernel key features

- ▶ Portability and hardware support. Runs on most architectures.
- ▶ Scalability. Can run on super computers as well as on tiny devices (4 MB of RAM is enough).
- ▶ Compliance to standards and interoperability.
- ▶ Exhaustive networking support.
- ▶ Security. It can't hide its flaws. Its code is reviewed by many experts.
- ▶ Stability and reliability.
- ▶ Modularity. Can include only what a system needs even at run time.
- ▶ Easy to program. You can learn from existing code. Many useful resources on the net.



Linux kernel in the system





Linux kernel main roles

- ▶ **Manage all the hardware resources:** CPU, memory, I/O.
- ▶ **Provide a set of portable, architecture and hardware independent APIs** to allow user space applications and libraries to use the hardware resources.
- ▶ **Handle concurrent accesses and usage** of hardware resources from different applications.
 - ▶ Example: a single network interface is used by multiple user space applications through various network connections. The kernel is responsible to “multiplex” the hardware resource.



System calls

- ▶ The main interface between the kernel and user space is the set of system calls
- ▶ About 400 system calls that provide the main kernel services
 - ▶ File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- ▶ This interface is stable over time: only new system calls can be added by the kernel developers
- ▶ This system call interface is wrapped by the C library, and user space applications usually never make a system call directly but rather use the corresponding C library function



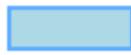
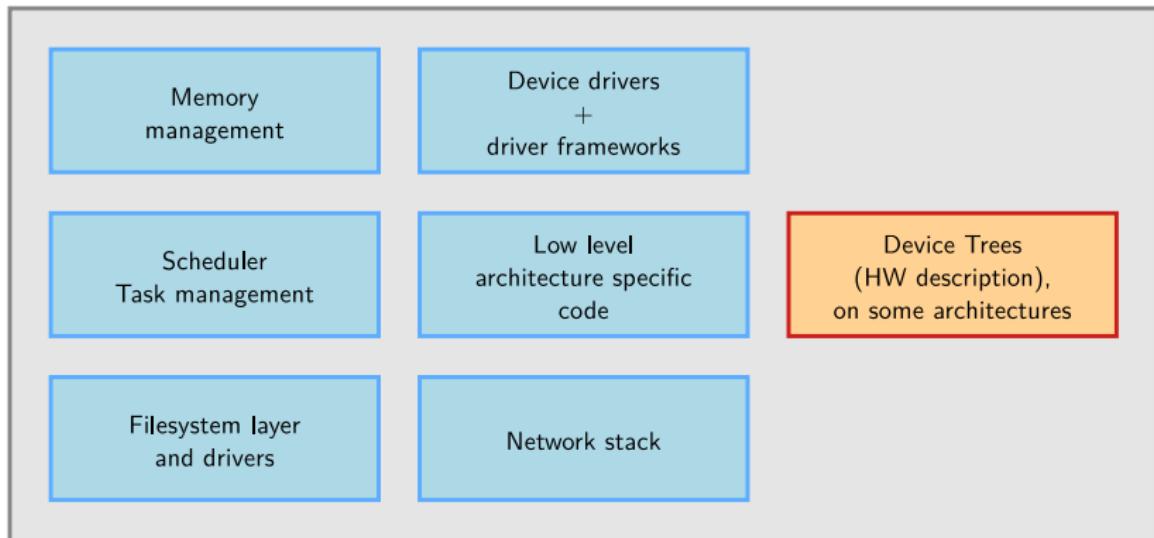
Pseudo filesystems

- ▶ Linux makes system and kernel information available in user space through **pseudo filesystems**, sometimes also called **virtual filesystems**
- ▶ Pseudo filesystems allow applications to see directories and files that do not exist on any real storage: they are created and updated on the fly by the kernel
- ▶ The two most important pseudo filesystems are
 - ▶ proc, usually mounted on /proc:
Operating system related information (processes, memory management parameters...)
 - ▶ sysfs, usually mounted on /sys:
Representation of the system as a set of devices and buses.
Information about these devices.



Inside the Linux kernel

Linux Kernel



Implemented mainly in C,
a little bit of assembly.



Written in a Device Tree
specific language.



Linux license

- ▶ The whole Linux sources are Free Software released under the GNU General Public License version 2 (GPL v2).
- ▶ For the Linux kernel, this basically implies that:
 - ▶ When you receive or buy a device with Linux on it, you should receive the Linux sources, with the right to study, modify and redistribute them.
 - ▶ When you produce Linux based devices, you must release the sources to the recipient, with the same rights, with no restriction.



Supported hardware architectures

- ▶ See the arch/ directory in the kernel sources
- ▶ Minimum: 32 bit processors, with or without MMU, and gcc support
- ▶ 32 bit architectures (arch/ subdirectories)
Examples: arm, arc, c6x, m68k, microblaze
- ▶ 64 bit architectures:
Examples: alpha, arm64, ia64...
- ▶ 32/64 bit architectures
Examples: mips, powerpc, riscv, sh, sparc, x86
- ▶ Find details in kernel sources: arch/<arch>/Kconfig,
arch/<arch>/README, or Documentation/<arch>/



Linux versioning scheme and development process

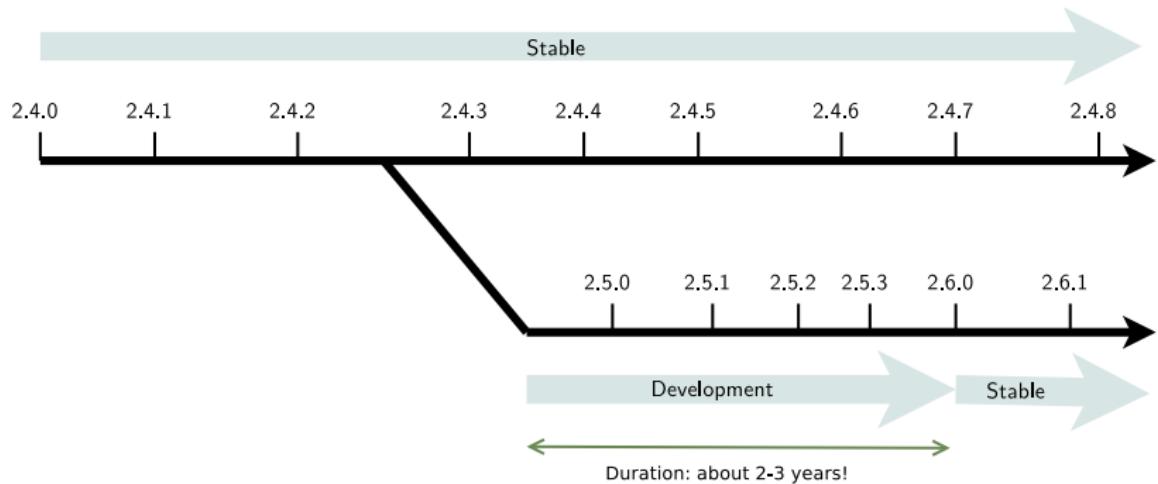


Until 2.6 (1)

- ▶ One stable major branch every 2 or 3 years
 - ▶ Identified by an even middle number
 - ▶ Examples: 1.0.x, 2.0.x, 2.2.x, 2.4.x
- ▶ One development branch to integrate new functionalities and major changes
 - ▶ Identified by an odd middle number
 - ▶ Examples: 2.1.x, 2.3.x, 2.5.x
 - ▶ After some time, a development version becomes the new base version for the stable branch
- ▶ Minor releases once in while: 2.2.23, 2.5.12, etc.



Until 2.6 (2)





Changes since Linux 2.6

- ▶ Since 2.6.0 (Dec. 2003), kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make disruptive changes to existing subsystems.
- ▶ Since then, there has been no need to create a new development branch massively breaking compatibility with the stable branch.
- ▶ Thanks to this, **more features are released to users at a faster pace.**



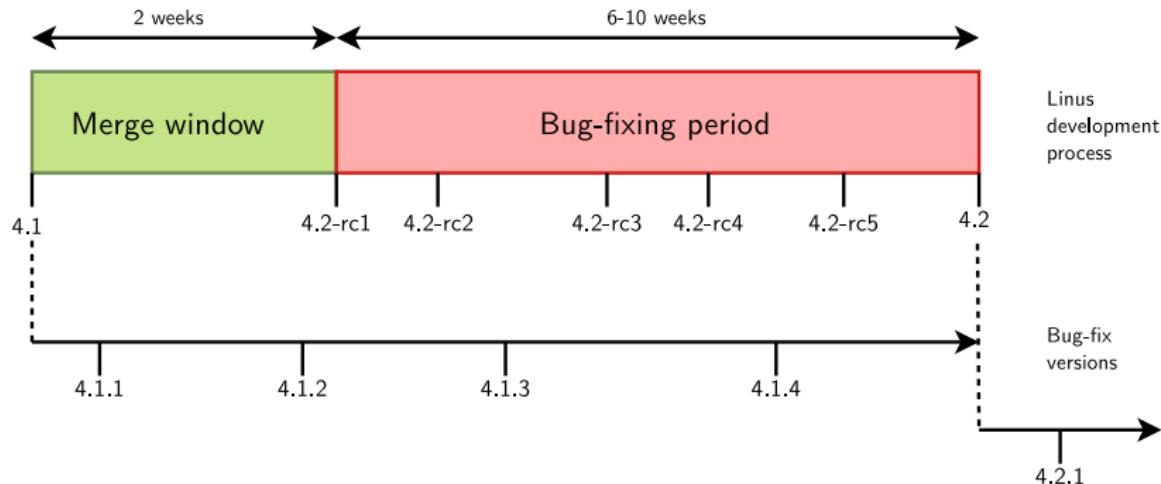
Versions since 2.6.0

- ▶ From 2003 to 2011, the official kernel versions were named 2.6.x.
- ▶ Linux 3.0 was released in July 2011
- ▶ Linux 4.0 was released in April 2015
- ▶ This is only a change to the numbering scheme
 - ▶ Official kernel versions are now named x.y (3.0, 3.1, 3.2, ..., 3.19, 4.0, 4.1, etc.)
 - ▶ Stabilized versions are named x.y.z (3.0.2, 4.2.7, etc.)
 - ▶ It effectively only removes a digit compared to the previous numbering scheme



New development model

Using merge and bug fixing windows





Need for long term support

- ▶ Issue: bug and security fixes only released for most recent stable kernel versions. Only LTS (*Long Term Support*) releases are supported for up to 6 years.
- ▶ Example at Google: starting from *Android O*, all new Android devices will have to run such an LTS kernel.
- ▶ You could also get long term support from a commercial embedded Linux provider.
- ▶ The *Civil Infrastructure Platform* project is an industry / Linux Foundation effort to support selected LTS versions (starting with 4.4) much longer (> 10 years). See <http://bit.ly/2hy1QYC>.

Longterm release kernels

Source: follow the "Releases" link on <http://kernel.org/>

Version	Maintainer	Released	Projected EOL
4.9	Greg Kroah-Hartman	2016-12-11	Jan, 2019
4.4	Greg Kroah-Hartman	2016-01-10	Feb, 2022
4.1	Sasha Levin	2015-06-21	Sep, 2017
3.16	Ben Hutchings	2014-08-03	Apr, 2020
3.10	Willy Tarreau	2013-06-30	Oct, 2017
3.2	Ben Hutchings	2012-01-04	May, 2018



What's new in each Linux release? (1)

The official list of changes for each Linux release is just a huge list of individual patches!

```
commit aa6e52a35d388e730f4df0ec2ec48294590cc459
Author: Thomas Petazzoni <thomas.petazzoni@bootlin.com>
Date:   Wed Jul 13 11:29:17 2011 +0200
```

at91: at91-ohci: support overcurrent notification

Several USB power switches (AIC1526 or MIC2026) have a digital output that is used to notify that an overcurrent situation is taking place. This digital outputs are typically connected to GPIO inputs of the processor and can be used to be notified of these overcurrent situations.

Therefore, we add a new overcurrent_pin[] array in the at91_usbh_data structure so that boards can tell the AT91 OHCI driver which pins are used for the overcurrent notification, and an overcurrent_supported boolean to tell the driver whether overcurrent is supported or not.

The code has been largely borrowed from ohci-da8xx.c and ohci-s3c2410.c.

Signed-off-by: Thomas Petazzoni <thomas.petazzoni@bootlin.com>
Signed-off-by: Nicolas Ferre <nicolas.ferre@atmel.com>

Very difficult to find out the key changes and to get the global picture out of individual changes.



What's new in each Linux release? (2)

Fortunately, there are some useful resources available

- ▶ <http://wiki.kernelnewbies.org/LinuxChanges>
(some versions are missing)
- ▶ <http://lwn.net>
- ▶ <http://www.linux-arm.info>
News about Linux on ARM, including kernel changes.
- ▶ <http://linuxfr.org>, for French readers



Building the kernel



Kernel configuration



Kernel configuration

- ▶ The kernel contains thousands of device drivers, filesystem drivers, network protocols and other configurable items
- ▶ Thousands of options are available, that are used to selectively compile parts of the kernel source code
- ▶ The kernel configuration is the process of defining the set of options with which you want your kernel to be compiled
- ▶ The set of options depends
 - ▶ On the target architecture and on your hardware (for device drivers, etc.)
 - ▶ On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.). Such generic options are available in all architectures.



Specifying the target architecture

First, specify the architecture for the kernel to build

- ▶ Set it to the name of a directory under `arch/`:
`export ARCH=arm`
- ▶ By default, the kernel build system assumes that the kernel is configured and built for the host architecture (`x86` in our case, native kernel compiling)
- ▶ The kernel build system will use this setting to:
 - ▶ Use the configuration options for the target architecture.
 - ▶ Compile the kernel with source code and headers for the target architecture.



Kernel configuration and build system

- ▶ The kernel configuration and build system is based on multiple Makefiles
- ▶ One only interacts with the main `Makefile`, present at the **top directory** of the kernel source tree
- ▶ Interaction takes place
 - ▶ using the `make` tool, which parses the `Makefile`
 - ▶ through various **targets**, defining which action should be done (configuration, compilation, installation, etc.). Run `make help` to see all available targets.
- ▶ Example
 - ▶ `cd linux-4.14.x/`
 - ▶ `make <target>`



Kernel configuration details

- ▶ The configuration is stored in the `.config` file at the root of kernel sources
 - ▶ Simple text file, key=value (included by the kernel Makefile)
- ▶ As options have dependencies, typically never edited by hand, but through graphical or text interfaces:
 - ▶ `make xconfig`, `make gconfig` (graphical)
 - ▶ `make menuconfig`, `make nconfig` (text)
 - ▶ You can switch from one to another, they all load/save the same `.config` file, and show the same set of options



Initial configuration

Difficult to find which kernel configuration will work with your hardware and root filesystem. Start with one that works!

- ▶ Desktop or server case:
 - ▶ Advisable to start with the configuration of your running kernel, usually available in /boot:
`cp /boot/config-`uname -r` .config`
- ▶ Embedded platform case:
 - ▶ Default configuration files are available, usually for each CPU family.
 - ▶ They are stored in `arch/<arch>/configs/`, and are just minimal `.config` files (only settings different from default ones).
 - ▶ Run `make help` to find if one is available for your platform
 - ▶ To load a default configuration file, just run
`make cpu_defconfig`
 - ▶ This will overwrite your existing `.config` file!

Now, you can make configuration changes (`make menuconfig...`).



Create your own default configuration

To create your own default configuration file:

- ▶ `make savedefconfig`

This creates a minimal configuration (non-default settings)

- ▶ `mv defconfig arch/<arch>/configs/myown_defconfig`
This way, you can share a reference configuration inside the kernel sources.



Kernel or module?

- ▶ The **kernel image** is a **single file**, resulting from the linking of all object files that correspond to features enabled in the configuration
 - ▶ This is the file that gets loaded in memory by the bootloader
 - ▶ All included features are therefore available as soon as the kernel starts, at a time where no filesystem exists
- ▶ Some features (device drivers, filesystems, etc.) can however be compiled as **modules**
 - ▶ These are *plugins* that can be loaded/unloaded dynamically to add/remove features to the kernel
 - ▶ Each **module is stored as a separate file in the filesystem**, and therefore access to a filesystem is mandatory to use modules
 - ▶ This is not possible in the early boot procedure of the kernel, because no filesystem is available



Kernel option types

There are different types of options

- ▶ `bool` options, they are either
 - ▶ `true` (to include the feature in the kernel) or
 - ▶ `false` (to exclude the feature from the kernel)
- ▶ `tristate` options, they are either
 - ▶ `true` (to include the feature in the kernel image) or
 - ▶ `module` (to include the feature as a kernel module) or
 - ▶ `false` (to exclude the feature)
- ▶ `int` options, to specify integer values
- ▶ `hex` options, to specify hexadecimal values
- ▶ `string` options, to specify string values



Kernel option dependencies

- ▶ There are dependencies between kernel options
- ▶ For example, enabling a network driver requires the network stack to be enabled
- ▶ Two types of dependencies
 - ▶ depends on dependencies. In this case, option A that depends on option B is not visible until option B is enabled
 - ▶ select dependencies. In this case, with option A depending on option B, when option A is enabled, option B is automatically enabled
- ▶ With the Show All Options option, `make xconfig` allows to see all options, even the ones that cannot be selected because of missing dependencies. Values for dependencies are shown.



make xconfig

make xconfig

- ▶ The most common graphical interface to configure the kernel.
- ▶ Make sure you read
help -> introduction: useful options!
- ▶ File browser: easier to load configuration files
- ▶ Search interface to look for parameters
- ▶ Required Debian / Ubuntu packages: qt5-default



make xconfig screenshot

Linux/arm 4.14.0 Kernel Configuration

File Edit Option Help

Option

General setup

- IRQ subsystem
- Timers subsystem
- CPU/Task time and stats accounting
- RCU Subsystem

Control Group support

- CPU controller
- Namespaces support
- Configure standard kernel features (expert users)
- Kernel Performance Events And Counters
 - GCC plugins
 - GCOV-based kernel profiling
 - Enable loadable module support
 - Enable the block layer
 - Partition Types
 - IO Schedulers

System Type

- Multiple platform selection
 - Marvell Engineering Business Unit (MVEBU) SoCs (NEW)
 - Actions Semi SoCs (NEW)
 - Axis Communications ARM based ARTPEC SoCs (NEW)
 - Atmel SoCs
 - Broadcom SoC Support (NEW)
 - Marvell Berlin SoCs (NEW)
 - Amlogic Meson SoCs (NEW)
 - Freescale i.MX Family (NEW)
 - Mediatek MT65xx & MT81xx SoC (NEW)
 - TI OMAP Common Features
- TI OMAP/AM/DM/DRA Family
 - TI OMAP2/3/4 Specific Features
 - Marvell PXA168/910/MMP2 (NEW)
 - Qualcomm Support (NEW)
 - ARM Ltd. RealView family (NEW)
 - Altera SOFCFGA family (NEW)

Option

Cross-compiler tool prefix:

- Compile also drivers which will not load
- Local version - append to kernel release:
- Automatically append version information to the version string

Kernel compression mode

- Gzip
- LZMA
- XZ
- LZO
- LZ4

Default hostname: (none)

- Support for paging of anonymous memory (swap)
- System V IPC
- POSIX Message Queues
- Enable process vm ready/writev syscalls

LZ4 (KERNEL_LZ4)

CONFIG_KERNEL_LZ4:

LZ4 is an LZ77-type compressor with a fixed, byte-oriented encoding. A preliminary version of LZ4 de/compression tool is available at <<https://code.google.com/p/lz4/>>.

Its compression ratio is worse than LZO. The size of the kernel is about 8% bigger than LZO. But the decompression speed is faster than LZO.

Symbol: KERNEL_LZ4 [=n]

Type : boolean

Prompt: LZ4

Location:

- General setup
- Kernel compression mode (<choice> [=y])

Defined at init/Kconfig:200

Depends on: <choice> && HAVE_KERNEL_LZ4 [=y]



make xconfig search interface

Looks for a keyword in the parameter name. Allows to select or unselect found parameters.

The screenshot shows the 'Search Config' dialog window. In the 'Find:' field, the text 'ftrace' is entered. Below the search bar, there is a list of kernel configuration options under the heading 'Option'. Several options are listed with checkboxes:

- Tracers
- enable/disable function tracing dynamically
- Perform a startup test on ftrace
- Trace syscalls
- Persistent function tracer
- Copy the output from kernel Ftrace to STM engine

Below the 'Option' section, a bolded heading 'Persistent function tracer (PSTORE_FTRACE)' is displayed. Underneath it, the text 'CONFIG_PSTORE_FTRACE:' is shown. A detailed description follows: 'With this option kernel traces function calls into a persistent ram buffer that can be decoded and dumped after reboot through pstore filesystem. It can be used to determine what function was last called before a reset or panic.' A note below says 'If unsure, say N.' At the bottom, the 'Symbol: PSTORE_FTRACE [=n]', 'Type : boolean', 'Prompt: Persistent function tracer', 'Location:', and 'Defined at fs/pstore/Kconfig:61' are listed, along with the dependency 'Depends on: MISC_FILESYSTEMS [=y] && PSTORE [=y] && FUNCTION_TRACER [=y] && DEBUG_FS [=y]'.



Kernel configuration options

Compiled as a module (separate file)

`CONFIG_ISO9660_FS=m`

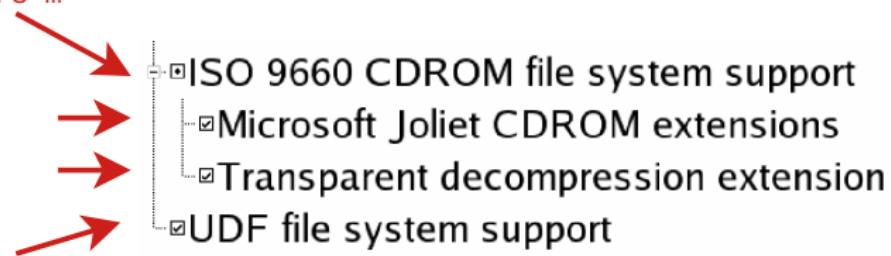
Driver options

`CONFIG_JOLIET=y`

`CONFIG_ZISOFS=y`

Compiled statically into the kernel

`CONFIG_UDF_FS=y`





Corresponding .config file excerpt

Options are grouped by sections and are prefixed with CONFIG_.

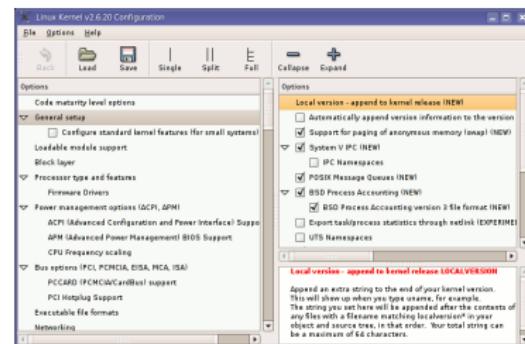
```
#  
# CD-ROM/DVD Filesystems  
#  
CONFIG_ISO9660_FS=m  
CONFIG_JOLIET=y  
CONFIG_ZISOFS=y  
CONFIG_UDF_FS=y  
CONFIG_UDF_NLS=y  
  
#  
# DOS/FAT/NT Filesystems  
#  
# CONFIG_MSDOS_FS is not set  
# CONFIG_VFAT_FS is not set  
CONFIG_NTFS_FS=m  
# CONFIG_NTFS_DEBUG is not set  
CONFIG_NTFS_RW=y
```



make gconfig

make gconfig

- ▶ GTK based graphical configuration interface.
Functionality similar to that of make xconfig.
- ▶ Just lacking a search functionality.
- ▶ Required Debian packages:
`libglade2-dev`

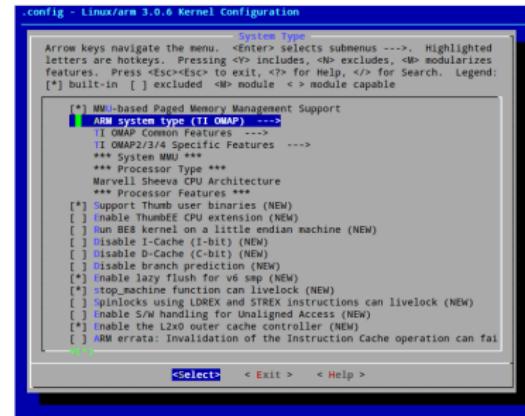




make menuconfig

make menuconfig

- ▶ Useful when no graphics are available. Pretty convenient too!
- ▶ Same interface found in other tools: BusyBox, Buildroot...
- ▶ Required Debian packages: libncurses-dev





make nconfig

make nconfig

- ▶ A newer, similar text interface
- ▶ More user friendly (for example, easier to access help information).
- ▶ Required Debian packages:
libncurses-dev

The screenshot shows the 'make nconfig' interface for a Linux/x86_64 3.0.0 Kernel Configuration. The menu tree is as follows:

- General setup ...>
- [] Enable loadable module support ...>
- * Enable the block layer ...>
- Processor type and features ...>
- Power management and ACPI options ...>
- Bus options (PCI etc.) ...>
- Executable file formats / Emulations ...>
- [] Networking support ...>
- Device Drivers ...>
- Firmware Drivers ...>
- File systems ...>
- Kernel hacking ...>
- Security options ...>
- [] Cryptographic API ...>
- [] Virtualization ...>
- Library routines ...>

At the bottom of the screen, there is a footer with the following keys:

F1 Help F2 Gym Info F3 Instal F4 Config F5 Back F6 Save F7 Load F8 Gym Search F9 Exit



make oldconfig

`make oldconfig`

- ▶ Needed very often!
- ▶ Useful to upgrade a `.config` file from an earlier kernel release
- ▶ Issues warnings for configuration parameters that no longer exist in the new kernel.
- ▶ Asks for values for new parameters (while `xconfig` and `menuconfig` silently set default values for new parameters).

If you edit a `.config` file by hand, it's strongly recommended to run `make oldconfig` afterwards!



Undoing configuration changes

A frequent problem:

- ▶ After changing several kernel configuration settings, your kernel no longer works.
- ▶ If you don't remember all the changes you made, you can get back to your previous configuration:
`$ cp .config.old .config`
- ▶ All the configuration interfaces of the kernel (xconfig, menuconfig, oldconfig...) keep this `.config.old` backup copy.



Compiling and installing the kernel



Choose a compiler

The compiler invoked by the kernel Makefile is
\$(CROSS_COMPILE)gcc

- ▶ When compiling natively
 - ▶ Leave CROSS_COMPILE undefined and the kernel will be natively compiled for the host architecture using gcc.
- ▶ When using a cross-compiler
 - ▶ To make the difference with a native compiler, cross-compiler executables are prefixed by the name of the target system, architecture and sometimes library. Examples:
`mips-linux-gcc`: the prefix is `mips-linux-`
`arm-linux-gnueabi-gcc`: the prefix is `arm-linux-gnueabi-`
 - ▶ So, you can specify your cross-compiler as follows:
`export CROSS_COMPILE=arm-linux-gnueabi-`

CROSS_COMPILE is actually the prefix of the cross compiling tools (gcc, as, ld, objcopy, strip...).



Specifying ARCH and CROSS_COMPILE

There are actually two ways of defining ARCH and CROSS_COMPILE:

- ▶ Pass ARCH and CROSS_COMPILE on the make command line:

```
make ARCH=arm CROSS_COMPILE=arm-linux-
```

Drawback: it is easy to forget to pass these variables when you run any `make` command, causing your build and configuration to be screwed up.

- ▶ Define ARCH and CROSS_COMPILE as environment variables:

```
export ARCH=arm
```

```
export CROSS_COMPILE=arm-linux-
```

Drawback: it only works inside the current shell or terminal.

You could put these settings in a file that you source every time you start working on the project. If you only work on a single architecture with always the same toolchain, you could even put these settings in your `~/.bashrc` file to make them permanent and visible from any terminal.



Kernel compilation

- ▶ make
 - ▶ In the main kernel source directory!
 - ▶ Remember to run multiple jobs in parallel if you have multiple CPU cores. Example: `make -j 4`
 - ▶ No need to run as root!
- ▶ Generates
 - ▶ `vmlinux`, the raw uncompressed kernel image, in the ELF format, useful for debugging purposes, but cannot be booted
 - ▶ `arch/<arch>/boot/*Image`, the final, usually compressed, kernel image that can be booted
 - ▶ `bzImage` for x86, `zImage` for ARM, `vmlinux.bin.gz` for ARC, etc.
 - ▶ `arch/<arch>/boot/dts/*.dtb`, compiled Device Tree files (on some architectures)
 - ▶ All kernel modules, spread over the kernel source tree, as `.ko` (*Kernel Object*) files.



Kernel installation: native case

- ▶ make install
 - ▶ Does the installation for the host system by default, so needs to be run as root.
- ▶ Installs
 - ▶ /boot/vmlinuz-<version>
Compressed kernel image. Same as the one in
arch/<arch>/boot
 - ▶ /boot/System.map-<version>
Stores kernel symbol addresses for debugging purposes
(obsolete: such information is usually stored in the kernel
itself)
 - ▶ /boot/config-<version>
Kernel configuration for this version
- ▶ In GNU/Linux distributions, typically re-runs the bootloader
configuration utility to make the new kernel available at the
next boot.



Kernel installation: embedded case

- ▶ `make install` is rarely used in embedded development, as the kernel image is a single file, easy to handle.
- ▶ Another reason is that there is no standard way to deploy and use the kernel image.
- ▶ Therefore making the kernel image available to the target is usually manual or done through scripts in build systems.
- ▶ It is however possible to customize the `make install` behaviour in `arch/<arch>/boot/install.sh`



Module installation: native case

- ▶ make modules_install
 - ▶ Does the installation for the host system by default, so needs to be run as root
- ▶ Installs all modules in /lib/modules/<version>/
 - ▶ kernel/
Module .ko (Kernel Object) files, in the same directory structure as in the sources.
 - ▶ modules.alias, modules.aliases.bin
Aliases for module loading utilities. Used to find drivers for devices. Example line:
`alias usb:v066Bp20F9d*dc*dsc*dp*ic*isc*ip*in* asix`
 - ▶ modules.dep, modules.dep.bin
Module dependencies
 - ▶ modules.symbols, modules.symbols.bin
Tells which module a given symbol belongs to.



Module installation: embedded case

- ▶ In embedded development, you can't directly use `make modules_install` as it would install target modules in `/lib/modules` on the host!
- ▶ The `INSTALL_MOD_PATH` variable is needed to generate the module related files and install the modules in the target root filesystem instead of your host root filesystem:
`make INSTALL_MOD_PATH=<dir>/ modules_install`



Kernel cleanup targets

- ▶ Clean-up generated files (to force re-compilation):
`make clean`
- ▶ Remove all generated files. Needed when switching from one architecture to another.
Caution: it also removes your `.config` file!
`make mrproper`
- ▶ Also remove editor backup and patch reject files (mainly to generate patches):
`make distclean`
- ▶ If you are in a git tree, remove all files not tracked (and ignored) by git:
`git clean -fdx`

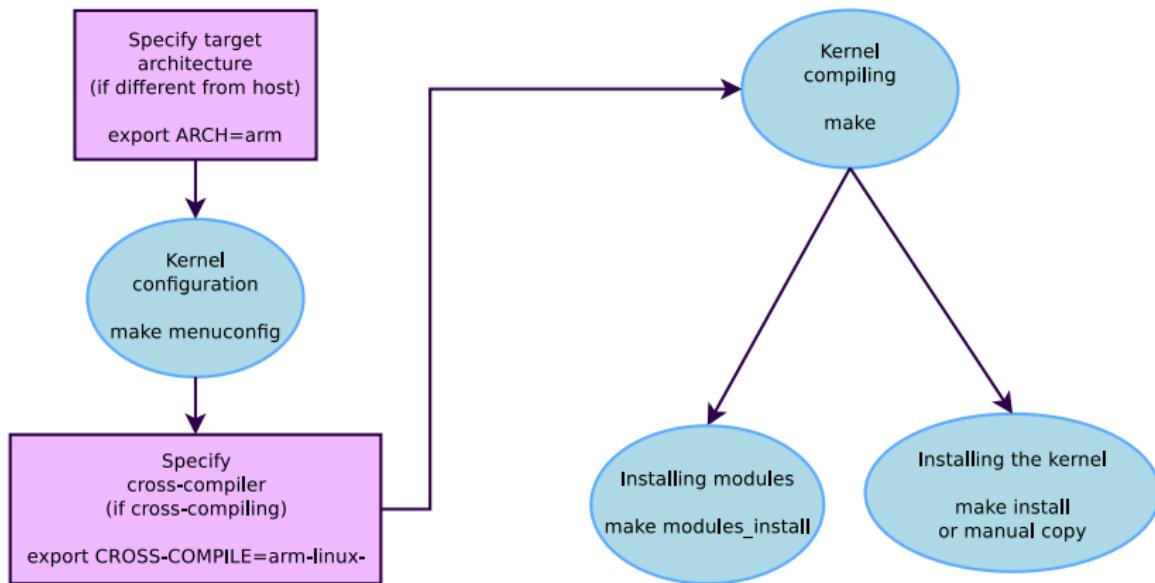




Kernel building overview

Environment setup
and configuration

Kernel building
and deployment





Booting the kernel



Device Tree (DT)

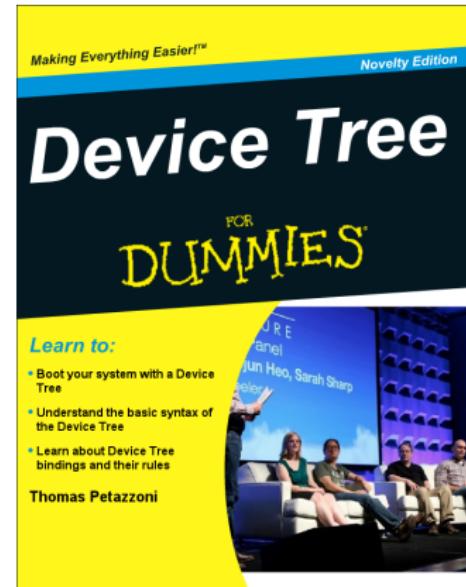
- ▶ Many embedded architectures have a lot of non-discoverable hardware.
- ▶ Depending on the architecture, such hardware is either described using C code directly within the kernel, or using a special hardware description language in a *Device Tree*.
- ▶ The DT was created for PowerPC, and later was adopted by other architectures (ARM, ARC...). Now Linux has DT support in most architectures, at least for specific systems (for example for the OLPC on x86).
- ▶ A *Device Tree Source*, written by kernel developers, is compiled into a binary *Device Tree Blob*, and needs to be passed to the kernel at boot time.
 - ▶ There is one different Device Tree for each board/platform supported by the kernel, available in `arch/arm/boot/dts/<board>.dtb`.
 - ▶ The bootloader must load both the kernel image and the Device Tree Blob in memory before starting the kernel.



Customize your board device tree!

Often needed for embedded board users:

- ▶ To describe external devices attached to non-discoverable busses (such as I2C) and configure them.
- ▶ To configure pin muxing: choosing what SoC signals are made available on the board external connectors.
- ▶ To configure some system parameters: flash partitions, kernel command line (other ways exist)
- ▶ Useful reference: Device Tree for Dummies, Thomas Petazzoni (Apr. 2014): <http://j.mp/1jQU6NR>





Booting with U-Boot

- ▶ Recent versions of U-Boot can boot the `zImage` binary.
- ▶ Older versions require a special kernel image format: `uImage`
 - ▶ `uImage` is generated from `zImage` using the `mkimage` tool. It is done automatically by the kernel `make uImage` target.
 - ▶ On some ARM platforms, `make uImage` requires passing a `LOADADDR` environment variable, which indicates at which physical memory address the kernel will be executed.
- ▶ In addition to the kernel image, U-Boot can also pass a *Device Tree Blob* to the kernel.
- ▶ The typical boot process is therefore:
 1. Load `zImage` or `uImage` at address X in memory
 2. Load `<board>.dtb` at address Y in memory
 3. Start the kernel with `bootz X - Y` (`zImage` case), or
`bootm X - Y` (`uImage` case)

The `-` in the middle indicates no *initramfs*



Kernel command line

- ▶ In addition to the compile time configuration, the kernel behaviour can be adjusted with no recompilation using the **kernel command line**
- ▶ The kernel command line is a string that defines various arguments to the kernel
 - ▶ It is very important for system configuration
 - ▶ `root=` for the root filesystem (covered later)
 - ▶ `console=` for the destination of kernel messages
 - ▶ Many more exist. The most important ones are documented in `admin-guide/kernel-parameters` in kernel documentation.
- ▶ This kernel command line is either
 - ▶ Passed by the bootloader. In U-Boot, the contents of the `bootargs` environment variable is automatically passed to the kernel
 - ▶ Specified in the Device Tree (for architectures which use it)
 - ▶ Built into the kernel, using the `CONFIG_CMDLINE` option.



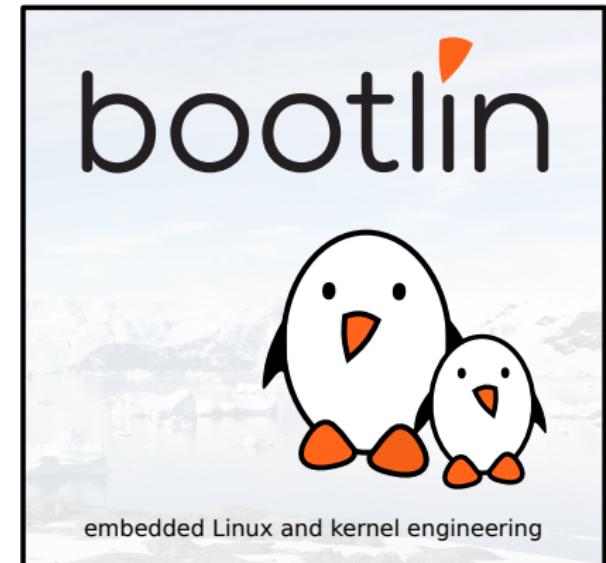
Practical lab - Compile and Boot an Android Kernel

- ▶ Extract the kernel patchset from Android Kernel
- ▶ Compile and boot a kernel for the emulator



Changes introduced in the Android Kernel

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Wakelocks



Power management basics

- ▶ Every CPU has a few states of power consumption, from being almost completely off, to working at full capacity.
- ▶ These different states are used by the Linux kernel to save power when the system is run
- ▶ For example, when the lid is closed on a laptop, it goes into “suspend”, which is the most power conservative mode of a device, where almost nothing but the RAM is kept awake
- ▶ While this is a good strategy for a laptop, it is not necessarily good for mobile devices
- ▶ For example, you don't want your music to be turned off when the screen is



Wakelocks

- ▶ Android's answer to these power management constraints is wakelocks
- ▶ One of the most famous Android changes, because of the flame wars it spawned
- ▶ The main idea is instead of letting the user decide when the devices need to go to sleep, the kernel is set to suspend as soon and as often as possible.
- ▶ In the same time, Android allows applications and kernel drivers to voluntarily prevent the system from going to suspend, keeping it awake (thus the name wakelock)
- ▶ This implies to write the applications and drivers to use the wakelock API.
 - ▶ Applications do so through the abstraction provided by the API
 - ▶ Drivers must do it themselves, which prevents to directly submit them to the vanilla kernel



Wakelocks API

► Kernel Space API

```
#include <linux/wakelock.h>
void wake_lock_init(struct wakelock *lock,
                     int type,
                     const char *name);
void wake_lock(struct wake_lock *lock);
void wake_unlock(struct wake_lock *lock);
void wake_lock_timeout(struct wake_lock *lock, long timeout);
void wake_lock_destroy(struct wake_lock *lock);
```

► User-Space API

```
$ echo foobar > /sys/power/wake_lock
$ echo foobar > /sys/power/wake_unlock
```



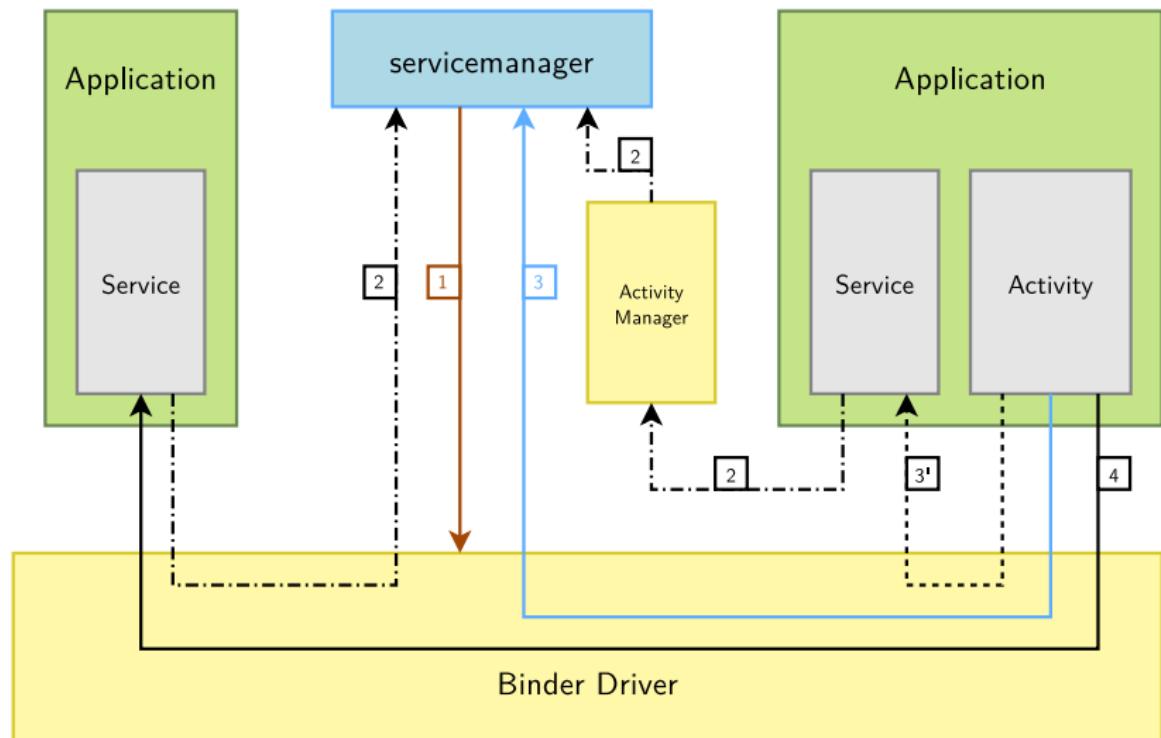
Binder



- ▶ RPC/IPC mechanism
- ▶ Takes its roots from BeOS and the OpenBinder project, which some of the current Android engineers worked on
- ▶ Adds remote object invocation capabilities to the Linux Kernel
- ▶ One of the very basic functionalities of Android. Without it, Android cannot work.
- ▶ Every call to the system servers go through Binder, just like every communication between applications, and even communication between the components of a single application.



Binder



→ `ioctl(BINDER_SET_CONTEXT_MGR)`
→ service registration

→ service lookup
→ service call
→ service direct call



klogger



Logging

- ▶ Logs are very important to debug a system, either live or after a fault occurred
- ▶ In a regular Linux distribution, two components are involved in the system's logging:
 - ▶ Linux' internal mechanism, accessible with the `dmesg` command and holding the output of all the calls to `printf()` from various parts of the kernel.
 - ▶ A syslog daemon, which handles the user space logs and usually stores them in the `/var/log` directory
- ▶ From Android developers' point of view, this approach has two flaws:
 - ▶ As the calls to `syslog()` go through as socket, they generate expensive task switches
 - ▶ Every call writes to a file, which probably writes to a slow storage device or to a storage device where writes are expensive



- ▶ Android addresses these issues with *logger*, which is a kernel driver, that uses 4 circular buffers in the kernel memory area.
- ▶ The buffers are exposed in the /dev/log directory and you can access them through the *liblog* library, which is in turn, used by the Android system and applications to write to logger, and by the *logcat* command to access them.
- ▶ This allows to have an extensive level of logging across the entire AOSP



Anonymous Shared Memory (ashmem)



Shared memory mechanism in Linux

- ▶ Shared memory is one of the standard IPC mechanisms present in most OSes
- ▶ Under Linux, they are usually provided by the POSIX SHM mechanism, which is part of the System V IPCs
- ▶ ndk/docs/system/libc/SYSV-IPC.html illustrates all the love Android developers have for these
- ▶ The bottom line is that they are flawed by design in Linux, and lead to code leaking resources, be it maliciously or not

- ▶ Ashmem is the response to these flaws
- ▶ Notable differences are:
 - ▶ Reference counting so that the kernel can reclaim resources which are no longer in use
 - ▶ There is also a mechanism in place to allow the kernel to shrink shared memory regions when the system is under memory pressure.
- ▶ The standard use of Ashmem in Android is that a process opens a shared memory region and share the obtained file descriptor through Binder.



Alarm Timers



The alarm driver

- ▶ Once again, the timer mechanisms available in Linux were not sufficient for the power management policy that Android was trying to set up
- ▶ High Resolution Timers can wake up a process, but don't fire when the system is suspended, while the Real Time Clock can wake up the system if it is suspended, but cannot wake up a particular process.
- ▶ Developed the alarm timers on top of the Real Time Clock and High Resolution Timers already available in the kernel
- ▶ These timers will be fired even if the system is suspended, waking up the device to do so
- ▶ Obviously, to let the application do its job, when the application is woken up, a wakelock is grabbed



Low Memory Killer



Low Memory Killer

- ▶ When the system goes out of memory, Linux throws the OOM Killer to cleanup memory greedy processes
- ▶ However, this behaviour is not predictable at all, and can kill very important components of a phone (Telephony stack, Graphic subsystem, etc) instead of low priority processes (Angry Birds)
- ▶ The main idea is to have another process killer, that kicks in before the OOM Killer and takes into account the time since the application was last used and the priority of the component for the system
- ▶ It uses various thresholds, so that it first notifies applications so that they can save their state, then begins to kill non-critical background processes, and then the foreground applications
- ▶ As it is run to free memory before the OOM Killer, the latter will never be run, as the system will never run out of memory



The ION Memory Allocator

- ▶ ION was introduced with Ice Cream Sandwich (4.0) version of Android
- ▶ Its role is to allocate memory in the system, for most of the possible cases, and to allow different devices to share buffers, without any copy, possibly from an user space application
- ▶ It's for example useful if you want to retrieve an image from a camera, and push it to the JPEG hardware encoder from an user space application
- ▶ The usual Linux memory allocators can only allocate a buffer that is up to 512 pages wide, with a page usually being 4kiB.
- ▶ There was previously for Android (and Linux in general) some vendor specific mechanism to allocate larger physically contiguous memory areas (`nvmap` for nVidia, `CMEM` for TI, etc.)



- ▶ ION is here to unify the interface to allocate memory in the system, no matter on which SoC you're running on.
- ▶ It uses a system of heaps, with Linux publishing the heaps available on a given system.
- ▶ By default, you have three different heaps:

system Memory virtually contiguous memory, backed by
 `vmalloc`

system contiguous Physically contiguous memory, backed by
 `kmalloc`

carveout Large physically contiguous memory,
 preallocated at boot

- ▶ It also has a user space interface so that processes can allocate memory to work on.
- ▶ <https://lwn.net/Articles/480055/>



Comparison with mainline equivalents

- ▶ ION has entered staging since 3.14. And:
 - ▶ The contiguous allocation of the buffers is done through CMA
 - ▶ The buffer sharing between devices is made through dma-buf
 - ▶ Its user space API also allows to allocate and share buffers from the user space, which was not possible otherwise.
 - ▶ This API is also used to set the allocation constraints devices might have (for example, when one particular device can only access a subset of the memory, or when it needs to setup an IOMMU)



Network Security



Paranoid Network

- ▶ In the standard Linux kernel, every application can open sockets and communicate over the Network
- ▶ However, Google was willing to apply a more strict policy with regard to network access
- ▶ Access to the network is a permission, with a per application granularity
- ▶ Filtered with the GID
- ▶ You need it to access IP, Bluetooth, raw sockets or RFCOMM



Various Drivers and Fixes



Various additions

- ▶ Android also has a lot of minor features added to the Linux kernel:
 - ▶ RAM Console, a RAM-based console that survives a reboot to hold kernel logs
 - ▶ *pmem*, a physically contiguous memory allocator, written specifically for the Qualcomm MSM SoCs. Obsolete Now.
 - ▶ ADB
 - ▶ YAFFS2
 - ▶ Timed GPIOs



Linux Mainline Patches Merge



History

- ▶ The Android Kernel patches were kept for a long time out of the official Linux release
- ▶ They were first integrated in 2.6.29, in `drivers/staging/android`
- ▶ They were then removed from the kernel 2.6.35, because Google was unwilling to help the mainlining process
- ▶ They were then added back in 3.3 (around 2 years later) and are still there at the time
- ▶ While Google did a great job at keeping most of their changes as isolated from the core as possible, making this easy to merge in the staging area, it wasn't true for the wakelocks, due to their invasive nature.



Wakelocks Support

- ▶ The kernel developers were not quite happy about the in-kernel APIs used by the wakelocks
- ▶ Due to the changes in every places of the kernel to state whether or not we were allowed to suspend, it was not possible to merge the changes as is: either you were getting all of it, or none
- ▶ Since version 3.5, two features were included in the kernel to implement opportunistic suspend:
 - `autosleep` is a way to let the kernel trigger suspend or hibernate whenever there are no active wakeup sources.
 - `wake locks` are a way to create and manipulate wakeup sources from user space. The interface is compatible with the android one.



Current State: Merged Patches

- ▶ As of 3.10, the following patches/features are now found in the mainline kernel:
 - ▶ Binder
 - ▶ Alarm Timers (under the name POSIX Alarm Timers introduced in 2.6.38)
 - ▶ Ashmem
 - ▶ Klogger
 - ▶ Timed GPIOs
 - ▶ Low Memory Killer
 - ▶ RAM Console (superseded by pstore RAM backend introduced in 3.5)



Current State: Missing Patches

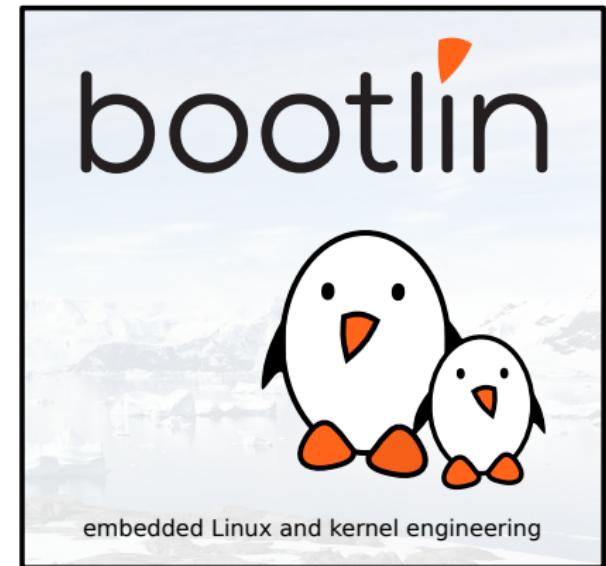
- ▶ As of 3.10, the following patches/features are missing from the mainline kernel:
 - ▶ Paranoid Networking
 - ▶ ION Memory Allocator
 - ▶ USB Gadget
 - ▶ FIQ debugger
 - ▶ pmem (removed in 3.3)



Android Bootloaders

Android Bootloaders

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Boot Sequence



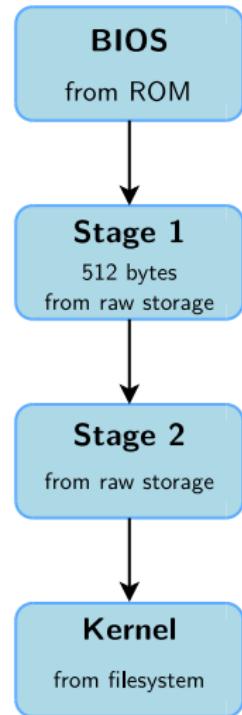
Bootloaders

- ▶ The bootloader is a piece of code responsible for
 - ▶ Basic hardware initialization
 - ▶ Loading of an application binary, usually an operating system kernel, from flash storage, from the network, or from another type of non-volatile storage.
 - ▶ Possibly decompression of the application binary
 - ▶ Execution of the application
- ▶ Besides these basic functions, most bootloaders provide a shell with various commands implementing different operations.
 - ▶ Loading of data from storage or network, memory inspection, hardware diagnostics and testing, etc.



Bootloaders on BIOS-based x86 (1)

- ▶ The x86 processors are typically bundled on a board with a non-volatile memory containing a program, the BIOS.
- ▶ On old BIOS-based x86 platforms: the BIOS is responsible for basic hardware initialization and loading of a very small piece of code from non-volatile storage.
- ▶ This piece of code is typically a 1st stage bootloader, which will load the full bootloader itself.
- ▶ It typically understands filesystem formats so that the kernel file can be loaded directly from a normal filesystem.
- ▶ This sequence is different for modern EFI-based systems.





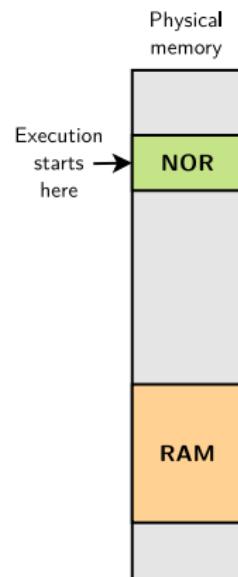
Bootloaders on x86 (2)

- ▶ GRUB, Grand Unified Bootloader, the most powerful one.
<http://www.gnu.org/software/grub/>
 - ▶ Can read many filesystem formats to load the kernel image and the configuration, provides a powerful shell with various commands, can load kernel images over the network, etc.
 - ▶ See our dedicated presentation for details:
<https://bootlin.com/doc/legacy/grub/>
- ▶ Syslinux, for network and removable media booting (USB key, CD-ROM)
<http://www.kernel.org/pub/linux/utils/boot/syslinux/>



Booting on embedded CPUs: case 1

- ▶ When powered, the CPU starts executing code at a fixed address
- ▶ There is no other booting mechanism provided by the CPU
- ▶ The hardware design must ensure that a NOR flash chip is wired so that it is accessible at the address at which the CPU starts executing instructions
- ▶ The first stage bootloader must be programmed at this address in the NOR
- ▶ NOR is mandatory, because it allows random access, which NAND doesn't allow
- ▶ **Not very common anymore** (unpractical, and requires NOR flash)



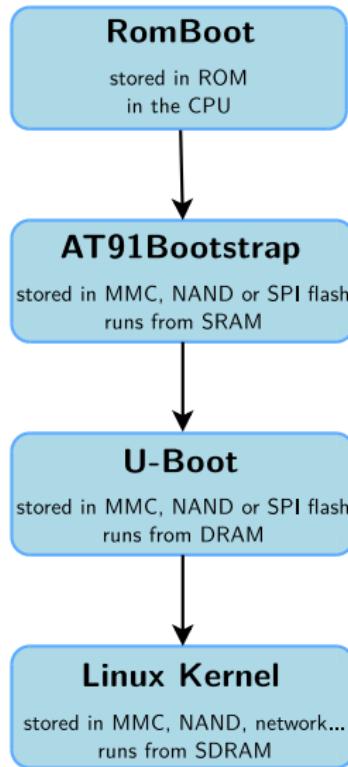


Booting on embedded CPUs: case 2

- ▶ The CPU has an integrated boot code in ROM
 - ▶ BootROM on AT91 CPUs, “ROM code” on OMAP, etc.
 - ▶ Exact details are CPU-dependent
- ▶ This boot code is able to load a first stage bootloader from a storage device into an internal SRAM (DRAM not initialized yet)
 - ▶ Storage device can typically be: MMC, NAND, SPI flash, UART (transmitting data over the serial line), etc.
- ▶ The first stage bootloader is
 - ▶ Limited in size due to hardware constraints (SRAM size)
 - ▶ Provided either by the CPU vendor or through community projects
- ▶ This first stage bootloader must initialize DRAM and other hardware devices and load a second stage bootloader into RAM



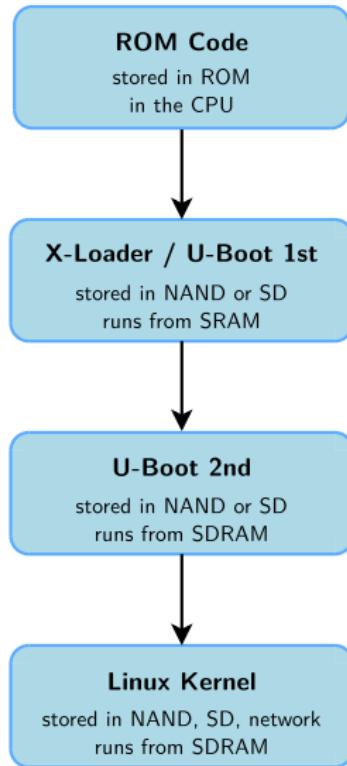
Booting on ARM Microchip AT91



- ▶ **RomBoot**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to 4 KB. No user interaction possible in standard boot mode.
- ▶ **AT91Bootstrap**: runs from SRAM. Initializes the DRAM, the NAND or SPI controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible.
- ▶ **U-Boot**: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided.
- ▶ **Linux Kernel**: runs from RAM. Takes over the system completely (the bootloader no longer exists).



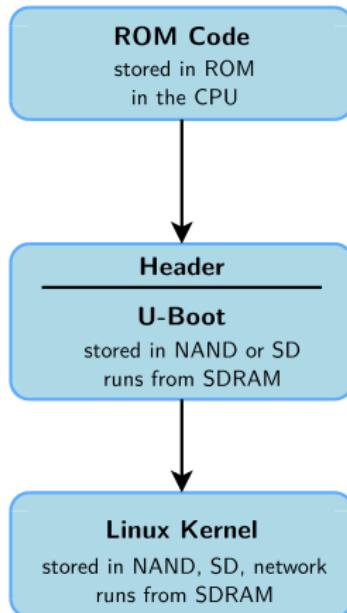
Booting on ARM TI OMAP3



- ▶ **ROM Code:** tries to find a valid bootstrap image from various storage sources, and load it into SRAM or RAM (RAM can be initialized by ROM code through a configuration header). Size limited to <64 KB. No user interaction possible.
- ▶ **X-Loader or U-Boot:** runs from SRAM. Initializes the DRAM, the NAND or MMC controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible. File called `MLO`.
- ▶ **U-Boot:** runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called `u-boot.bin` or `u-boot.img`.
- ▶ **Linux Kernel:** runs from RAM. Takes over the system completely (bootloaders no longer exists).



Booting on Marvell SoC



- ▶ **ROM Code:** tries to find a valid bootstrap image from various storage sources, and load it into RAM. The RAM configuration is described in a CPU-specific header, prepended to the bootloader image.
- ▶ **U-Boot:** runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called `u-boot.kwb`.
- ▶ **Linux Kernel:** runs from RAM. Takes over the system completely (bootloaders no longer exists).



Generic bootloaders for embedded CPUs

- ▶ We will focus on the generic part, the main bootloader, offering the most important features.
- ▶ There are several open-source generic bootloaders. Here are the most popular ones:
 - ▶ **U-Boot**, the universal bootloader by Denx
The most used on ARM, also used on PPC, MIPS, x86, m68k, NIOS, etc. The de-facto standard nowadays. We will study it in detail.
<http://www.denx.de/wiki/U-Boot>
 - ▶ **Barebox**, an architecture-neutral bootloader, written as a successor of U-Boot. It doesn't have as much hardware support as U-Boot yet. U-Boot has improved quite a lot thanks to this competitor.
<http://www.barebox.org>
- ▶ There are also a lot of other open-source or proprietary bootloaders, often architecture-specific
 - ▶ RedBoot, Yaboot, PMON, etc.



Fastboot



Definition

- ▶ Fastboot is a protocol to communicate with bootloaders over USB
- ▶ It is very simple to implement, making it easy to port on both new devices and on host systems
- ▶ Accessible with the `fastboot` command

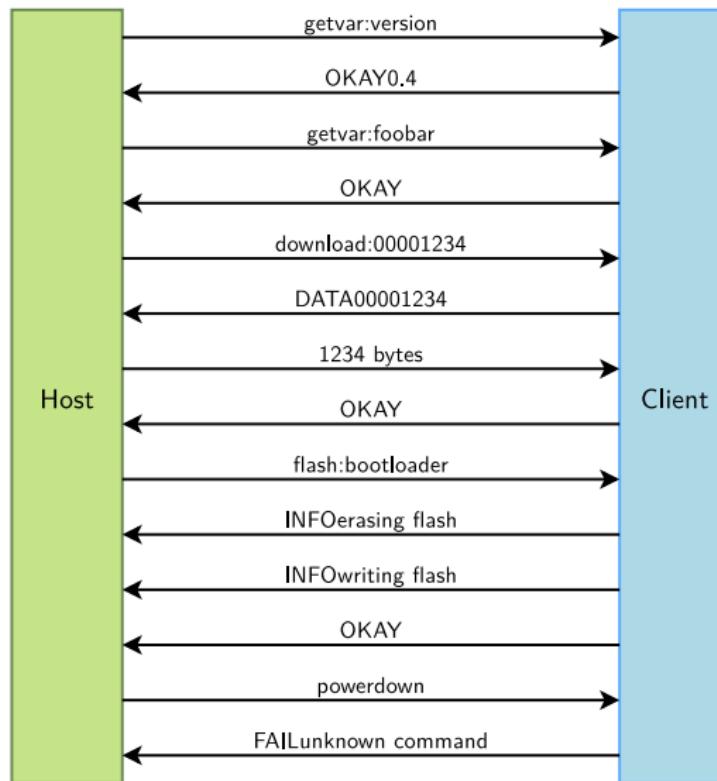


The Fastboot protocol

- ▶ It is very restricted, only 10 commands are defined in the protocol specifications
- ▶ It is synchronous and driven by the host
- ▶ Allows to:
 - ▶ Transmit data
 - ▶ Flash the various partitions of the device
 - ▶ Get variables from the bootloader
 - ▶ Control the boot sequence



Session example





Booting into Fastboot

- ▶ On most devices, it's disabled by default (the bootloader won't even implement it)
- ▶ On devices that support it, such as Google Nexus', you have several options:
 - ▶ Use a combination of keys at boot to start the bootloader right away into its fastboot mode
 - ▶ Use the `adb reboot bootloader` command on your workstation. The device will reboot in fastboot mode, awaiting for inputs.
- ▶ You can then interact with the device through the `fastboot` command on your workstation



Major Fastboot Commands

- ▶ You can get all the commands through `fastboot -h`
- ▶ The most widely used commands are:
 - `devices` Lists the fastboot-capable devices
 - `boot` Downloads a kernel and boots on it
 - `erase` Erases a given flash partition name
 - `flash` Writes a given file to a given flash partition
 - `getvar` Retrieves a variable from the bootloader
 - `continue` Goes on with a regular boot



getvar Variables

- ▶ Vendor-specific variables must also begin with a upper-case letter. Variables beginning with a lower-case letter are reserved for the Fastboot specifications and their evolution.

`version` Version of the Fastboot protocol implemented

`version-bootloader` Version of the bootloader

`version-baseband` Version of the baseband firmware

`product` Name of the product

`serialno` Product serial number

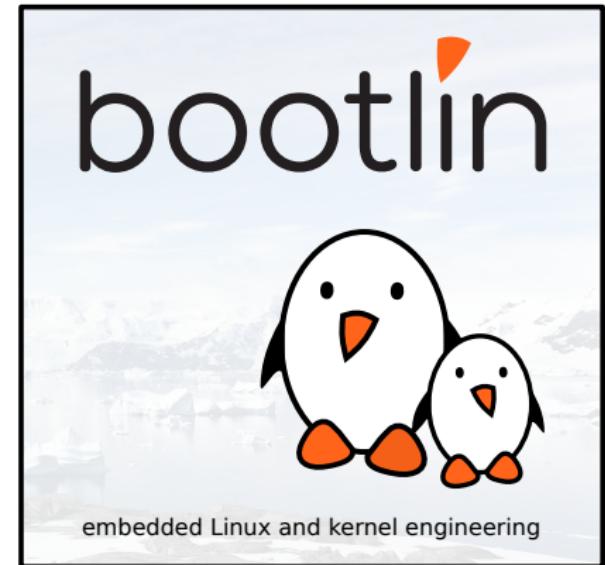
`secure` Does the bootloader require signed images?



Android Build System: Basics

Android Build System: Basics

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Android Build System: Basics

Basics



Build Systems

- ▶ Build systems are designed to meet several goals:
 - ▶ Integrate all the software components, both third-party and in-house into a working image
 - ▶ Be able to easily reproduce a given build
- ▶ Usually, they build software using the existing building system shipped with each component
- ▶ Several solutions: *Yocto*, *Buildroot*, *ptxdist*.
- ▶ Google came up with its own solution for Android, that never relies on other build systems, except for *GNU/Make*
 - ▶ It allows to rely on very few tools, and to control every software component in a consistent way.
 - ▶ But it also means that when you have to import a new component, you have to rewrite the whole Makefile to build it



First compilation

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

```
$ lunch
```

You're building on Linux

Lunch menu... pick a combo:

1. generic-eng
2. simulator
3. full_passion-userdebug
4. full_crespo-userdebug

Which would you like? [generic-eng]

```
$ make
```

```
$ make showcommands
```



Android Build System: Basics

envsetup.sh



Purpose

- ▶ Obviously modifies the current environment, that's why we have to source it
- ▶ It adds many useful shell macros
- ▶ These macros will serve several purposes:
 - ▶ Configure and set up the build system
 - ▶ Ease the navigation in the source code
 - ▶ Ease the development process
- ▶ Some macros will modify the environment variables, to be used by the build system later on



Environments variables exported 1/2

- ▶ ANDROID_EABI_TOOLCHAIN
 - ▶ Path to the Android prebuilt toolchain (`.../prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin`)
- ▶ ANDROID_TOOLCHAIN
 - ▶ Equals to ANDROID_EABI_TOOLCHAIN
- ▶ ANDROID_QTOOLS
 - ▶ Tracing tools for qemu (`.../development/emulator/qtools`).
This is weird however, since this path doesn't exist at all
- ▶ ANDROID_BUILD_PATHS
 - ▶ Path containing all the folders containing tools for the build
(`.../out/host/linux-x86/bin:$ANDROID_TOOLCHAIN:$ANDROID_QTOOLS:$ANDROID_TOOLCHAIN:$ANDROID_EABI_TOOLCHAIN`)



Environments variables exported 2/2

- ▶ `JAVA_HOME`
 - ▶ Path to the Java environment (`/usr/lib/jvm/java-6-sun`)
- ▶ `ANDROID_JAVA_TOOLCHAIN`
 - ▶ Path to the Java toolchain (`$JAVA_HOME/bin`)
- ▶ `ANDROID_PRE_BUILD_PATHS`
 - ▶ Alias to `ANDROID_JAVA_TOOLCHAIN`
- ▶ `ANDROID_PRODUCT_OUT`
 - ▶ Path to where the generated files will be for this product
(`.../out/target/product/<product_name>`)
- ▶ `OUT`
 - ▶ Alias to `ANDROID_PRODUCT_OUT`



Defined Commands 1/2

- lunch** Used to configure the build system
- croot** Changes the directory to go back to the root of the Android source tree
- cproj** Changes the directory to go back to the root of the current package
- tapas** Configure the build system to build a given application
- m** Makes the whole build from any directory in the source tree
- mm** Builds the modules defined in the current directory
- mmm** Builds the modules defined in the given directory



Defined Commands 2/2

`cgrep` Greps the given pattern on all the C/C++/header files

`jgrep` Greps the given pattern on all the Java files

`resgrep` Greps the given pattern on all the resources files

`mgrep` Greps the given pattern on all the Makefiles

`sgrep` Greps the given pattern on all Android source file

`godir` Go to the directory containing the given file

`pid` Use ADB to get the PID of the given process

`gdbclient` Use ADB to set up a remote debugging session

`key_back` Sends a input event corresponding to the Back key to the device



Configuration of the Build System



Configuration

- ▶ The Android build system is not much configurable compared to other build systems, but it is possible to modify to some extent
- ▶ Among the several configuration options you have, you can add extra flags for the C compiler, have a given package built with debug options, specify the output directory, and first of all, choose what product you want to build.
- ▶ This is done either through the `lunch` command or through a `buildspec.mk` file placed at the top of the source directory



lunch

- ▶ lunch is a shell function defined in build/envsetup.sh
- ▶ It is the easiest way to configure a build. You can either launch it without any argument and it will ask to choose among a list of known “combos” or launch it with the desired combos as argument.
- ▶ It sets the environment variables needed for the build and allows to start compiling at last
- ▶ You can declare new combos through the add_lunch_combo command
- ▶ These combos are the aggregation of the product to build and the variant to use (basically, which set of modules to install)



Variables Exported by Lunch

- ▶ TARGET_PRODUCT
 - ▶ Which product to build. To build for the emulator, you will have `aosp_<arch>`
- ▶ TARGET_BUILD_VARIANT
 - ▶ Select which set of modules to build, among
 - ▶ `user`: Includes modules tagged `user` (`Phone`)
 - ▶ `userdebug`: Includes modules tagged `user` or `debug` (`strace`)
 - ▶ `eng`: Includes modules tagged `user`, `debug` or `eng`:
(`e2fsprogs`)
- ▶ TARGET_BUILD_TYPE
 - ▶ Either `release` or `debug`. If `debug` is set, it will enable some debug options across the whole system.



buildspec.mk

- ▶ While `lunch` is convenient to quickly switch from one configuration to another. If you have only one product or you want to do more fine-grained configuration, this is not really convenient
- ▶ The file `buildspec.mk` is here for that.
- ▶ If you place it at the top of the sources, it will be used by the build system to get its configuration instead of relying on the environment variables
- ▶ It offers more variables to modify, such as compiling a given module with debugging symbols, additional C compiler flags, change the output directory...
- ▶ A sample is available in `build/buildspec.mk.default`, with lots of comments on the various variables.



Android Build System: Basics

Results



Output

- ▶ All the output is generated in the `out/` directory, outside of the source code directory
- ▶ This directory contains mostly two subdirectories: `host/` and `target/`
- ▶ These directories contain all the objects files compiled during the build process: `.o` files for C/C++ code, `.jar` files for Java libraries, etc
- ▶ It is an interesting feature, since it keeps all the generated stuff separate from the source code, and we can easily clean without side effects



Images

- ▶ It also generates the system images in the `out/target/product/<device_name>/` directory
- ▶ These images are:
 - `boot.img` A basic Android image, containing only the needed components to boot: a kernel image and a minimal system
 - `system.img` The remaining parts of Android. Much bigger, it contains most of the framework, applications and daemons
 - `userdata.img` A partition that will hold the user generated content. Mostly empty at compilation.
 - `recovery.img` A recovery image that allows to be able to debug or restore the system when something nasty happened.

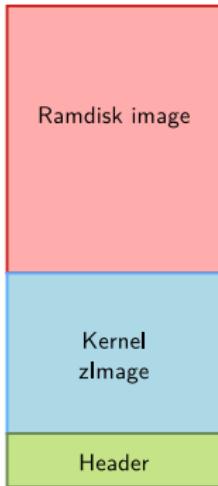


Android Boot Images

- ▶ The boot images are actually an Android-specific format, that holds most of what the bootloader expects
- ▶ They contain useful information, like the kernel command line, where to load the kernel, but also the image of the kernel, and an optional initramfs image
- ▶ A custom `mkbootimg` tool is used by Android to generate these images at compilation time from the kernel and the system it's generating
- ▶ We can tweak the behaviour of that tool from the build system configuration, that allows a great flexibility



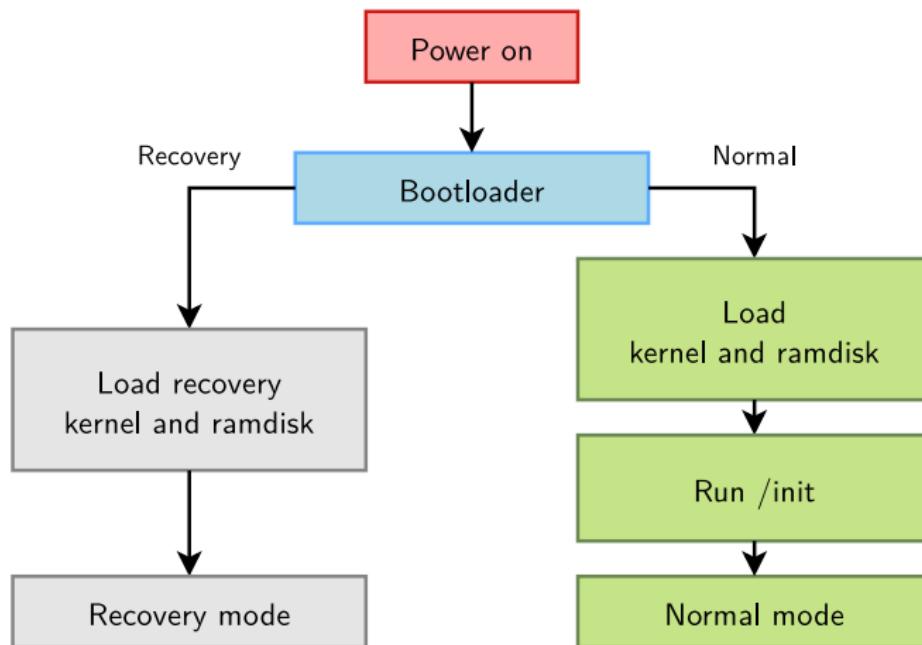
Android boot and recovery images



```
struct boot_img_hdr {  
    unsigned char magic[8]; /* "ANDROID!" */  
    unsigned kernel_size; /* size in bytes */  
    unsigned kernel_addr; /* physical load addr */  
  
    unsigned ramdisk_size; /* size in bytes */  
    unsigned ramdisk_addr; /* physical load addr */  
  
    unsigned second_size; /* size in bytes */  
    unsigned second_addr; /* physical load addr */  
  
    unsigned tags_addr; /* physical addr for kernel tags */  
    unsigned page_size; /* flash page size we assume, usually 2048 */  
    unsigned unused[2]; /* future expansion: should be 0 */  
  
    unsigned char name[16]; /* ascii product name */  
    unsigned char cmdline[512];  
  
    unsigned id[8]; /* timestamp / checksum / sha1 / etc */  
};  
  
from system/core/mkbootimg/bootimg.h
```



Boot sequence





Cleaning

- ▶ Cleaning is almost as easy as `rm -rf out/`
- ▶ `make clean` or `make clobber` deletes all generated files.
- ▶ `make installclean` removes the installed files for the current combo. It is useful when you work with several products to avoid doing a full rebuild each time you change from one to the other



Practical lab - Supporting a New Board

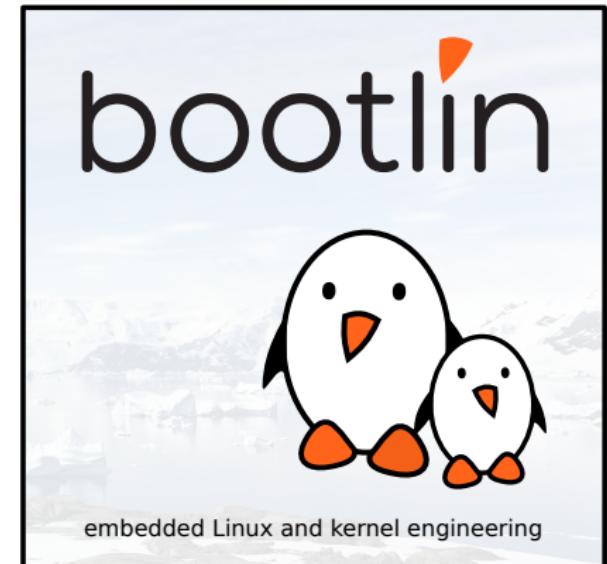


- ▶ Boot Android on a real hardware
- ▶ Troubleshoot simple problems on Android
- ▶ Generate a working build



Developing and Debugging with ADB

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Introduction



ADB

- ▶ Usually on embedded devices, debugging is done either through a serial port on the device or JTAG for low-level debugging
- ▶ This setup works well when developing a new product that will have a static system. You develop and debug a system on a product with serial and JTAG ports, and remove these ports from the final product.
- ▶ For mobile devices, where you will have applications developers that are not in-house, this is not enough.
- ▶ To address that issue, Google developed ADB, that runs on top of USB, so that another developer can still have debugging and low-level interaction with a production device.

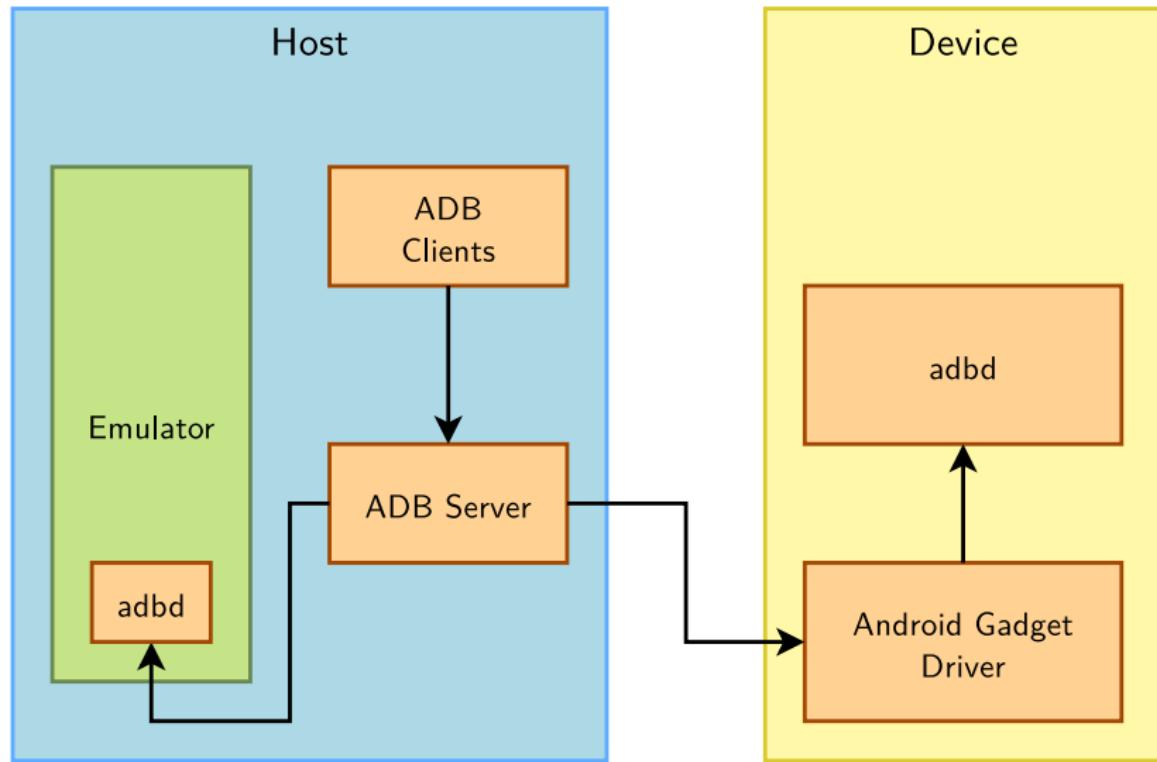


Implementation

- ▶ The code is split in 3 components:
 - ▶ ADBd, the part that runs on the device
 - ▶ ADB server, which is run on the host, acts as a proxy and manages the connection to ADBd
 - ▶ ADB clients, which are also run on the host, and are what is used to send commands to the device
- ▶ ADBd can work either on top of TCP or USB.
 - ▶ For USB, Google has implemented a driver using the USB gadget and the USB composite frameworks as it implements either the ADB protocol and the USB Mass Storage mechanism.
 - ▶ For TCP, ADBd just opens a socket
- ▶ ADB can also be used as a transport layer between the development platform and the device, disregarding whether it uses USB or TCP as underneath layer



ADB Architecture





Use of ADB



ADB commands: Basics

`start-server` Starts the ADB server on the host

`kill-server` Kills the ADB server on the host

`devices` Lists accessible devices

`connect` Connects to a remote ADBd using TCP port 5555 by default

`disconnect` Disconnects from a connected device

`help` Prints available commands with help information

`version` Prints the version number



ADB commands: Files and applications

push Copies a local file to the device

pull Copies a remote file from the device

sync There are three cases here:

- ▶ If no argument is passed, copies the local directories `system` and `data` if they differ from `/system` and `/data` on the target.
- ▶ If either `system` or `data` is passed, syncs this directory with the associated partition on the device
- ▶ Else, syncs the given folder

install Installs the given Android package (apk) on the device

uninstall Uninstalls the given package name from the device



ADB commands: Debugging

`logcat` Prints the device logs. You can filter either on the source of the logs or their on their priority level

`shell` Runs a remote shell with a command line interface. If an argument is given, runs it as a command and prints out the result

`bugreport` Gets all the relevant information to generate a bug report from the device: logs, internal state of the device, etc.

`jdwp` Lists the processes that support the JDWP protocol



ADB commands: Scripting 1/2

wait-for-device Blocks until the device gets connected to ADB.

You can also add additional commands to be run when the device becomes available.

get-state Prints the current state of the device, offline, bootloader or device

get-serialno Prints the serial number of the device

remount Remounts the `/system` partition on the device in read/write mode



ADB commands: Scripting 2/2

reboot Reboots the device. bootloader and recovery arguments are available to select the operation mode you want to reboot to.

reboot-bootloader Reboots the device into the bootloader

root Restarts ADBd with root permissions on the device

- ▶ Useful if the ro.secure property is set to 1 to force ADB into user mode. But ro.debuggable has to be set to 1 to allow to restart ADB as root

usb Restarts ADBd listening on USB

tcpip Restarts ADBd listening on TCP on the given port



ADB commands: Easter eggs

lolcat Alias to adb logcat

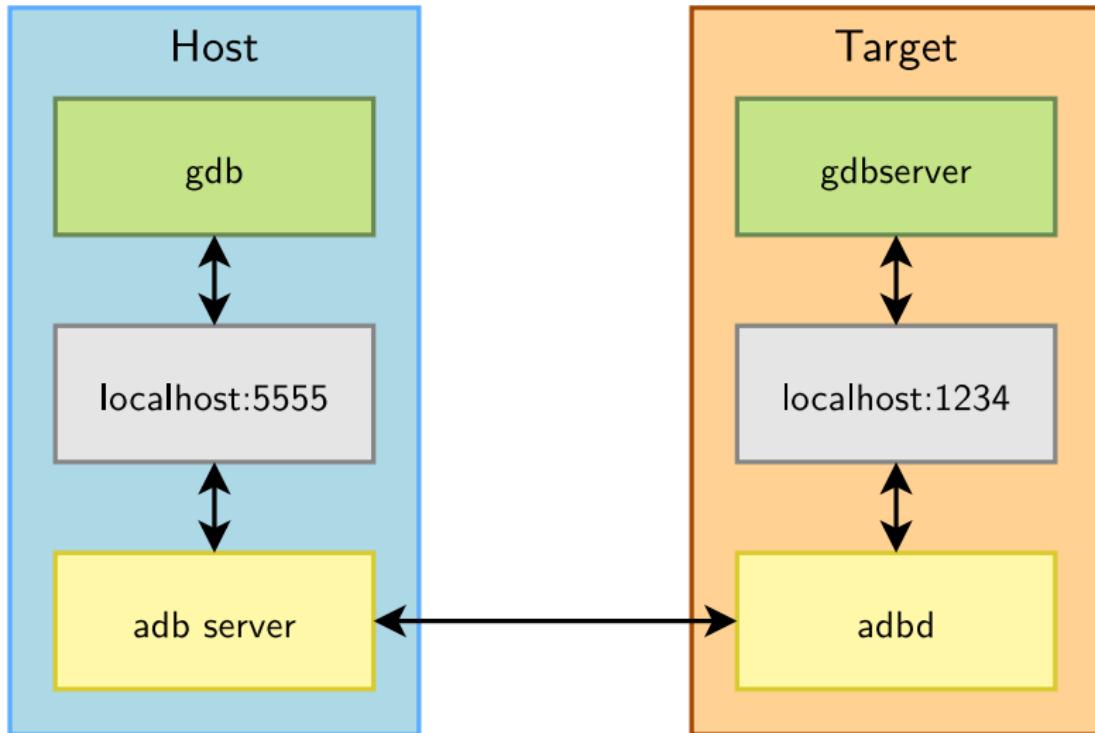
hell Equivalent to adb shell, with a different color scheme



Examples



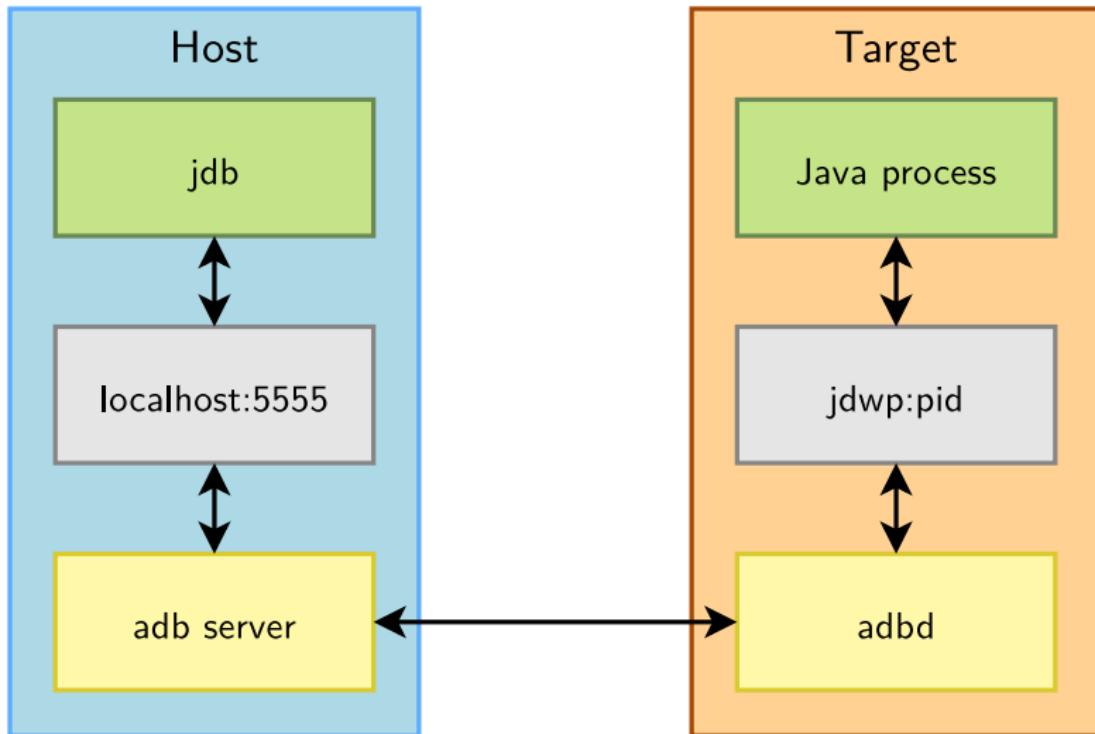
ADB forward and gdb



adb forward tcp:5555 tcp:1234
See also gdbclient



ADB forward and jdb



adb forward tcp:5555 jdwp:4242

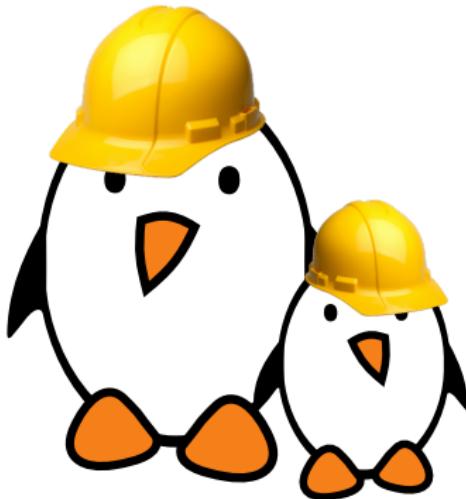


Various commands

- ▶ Wait for a device and install an application
 - ▶ adb wait-for-device install foobar.apk
- ▶ Test an application by sending random user input
 - ▶ adb shell monkey -v -p com.bootlin.foobar 500
- ▶ Filter system logs
 - ▶ adb logcat ActivityManager:I FooBar:D *:S
 - ▶ You can also set the ANDROID_LOG_TAGS environment variable on your workstation
- ▶ Access other log buffers
 - ▶ adb logcat -b radio



Practical lab - Use ADB



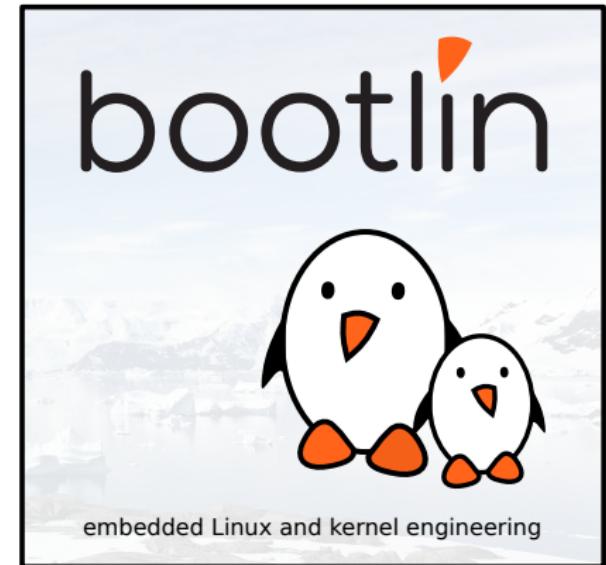
- ▶ Debug your system and applications
- ▶ Get a shell on a device
- ▶ Exchange files with a device
- ▶ Install new applications



Android Filesystem

Android Filesystem

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Principle and solutions



Filesystems

- ▶ Filesystems are used to organize data in directories and files on storage devices or on the network. The directories and files are organized as a hierarchy
- ▶ In Unix systems, applications and users see a **single global hierarchy** of files and directories, which can be composed of several filesystems.
- ▶ Filesystems are **mounted** in a specific location in this hierarchy of directories
 - ▶ When a filesystem is mounted in a directory (called *mount point*), the contents of this directory reflects the contents of the storage device
 - ▶ When the filesystem is unmounted, the *mount point* is empty again.
- ▶ This allows applications to access files and directories easily, regardless of their exact storage location



Filesystems (2)

- ▶ Create a mount point, which is just a directory

```
$ mkdir /mnt/usbkey
```

- ▶ It is empty

```
$ ls /mnt/usbkey  
$
```

- ▶ Mount a storage device in this mount point

```
$ mount -t vfat /dev/sda1 /mnt/usbkey  
$
```

- ▶ You can access the contents of the USB key

```
$ ls /mnt/usbkey  
docs prog.c picture.png movie.avi  
$
```



mount / umount

- ▶ mount allows to mount filesystems
 - ▶ `mount -t type device mountpoint`
 - ▶ `type` is the type of filesystem
 - ▶ `device` is the storage device, or network location to mount
 - ▶ `mountpoint` is the directory where files of the storage device or network location will be accessible
 - ▶ `mount` with no arguments shows the currently mounted filesystems
- ▶ umount allows to unmount filesystems
 - ▶ This is needed before rebooting, or before unplugging a USB key, because the Linux kernel caches writes in memory to increase performance. `umount` makes sure that these writes are committed to the storage.



Root filesystem

- ▶ A particular filesystem is mounted at the root of the hierarchy, identified by /
- ▶ This filesystem is called the **root filesystem**
- ▶ As `mount` and `umount` are programs, they are files inside a filesystem.
 - ▶ They are not accessible before mounting at least one filesystem.
- ▶ As the root filesystem is the first mounted filesystem, it cannot be mounted with the normal `mount` command
- ▶ It is mounted directly by the kernel, according to the `root=kernel` option
- ▶ When no root filesystem is available, the kernel panics

Please append a correct "root=" boot option

Kernel panic - not syncing: VFS: Unable to mount root fs on unknown block(0,0)



Location of the root filesystem

- ▶ It can be mounted from different locations
 - ▶ From the partition of a hard disk
 - ▶ From the partition of a USB key
 - ▶ From the partition of an SD card
 - ▶ From the partition of a NAND flash chip or similar type of storage device
 - ▶ From the network, using the NFS protocol
 - ▶ From memory, using a pre-loaded filesystem (by the bootloader)
 - ▶ etc.
- ▶ It is up to the system designer to choose the configuration for the system, and configure the kernel behaviour with `root=`



Mounting rootfs from storage devices

- ▶ Partitions of a hard disk or USB key
 - ▶ `root=/dev/sdXY`, where X is a letter indicating the device, and Y a number indicating the partition
 - ▶ `/dev/sdb2` is the second partition of the second disk drive (either USB key or ATA hard drive)
- ▶ Partitions of an SD card
 - ▶ `root=/dev/mmcblkXpY`, where X is a number indicating the device and Y a number indicating the partition
 - ▶ `/dev/mmcblk0p2` is the second partition of the first device
- ▶ Partitions of flash storage
 - ▶ `root=/dev/mtdblockX`, where X is the partition number
 - ▶ `/dev/mtdblock3` is the fourth partition of a NAND flash chip (if only one NAND flash chip is present)



rootfs in memory: initramfs (1)

- ▶ It is also possible to have the root filesystem integrated into the kernel image
- ▶ It is therefore loaded into memory together with the kernel
- ▶ This mechanism is called **initramfs**
 - ▶ It integrates a compressed archive of the filesystem into the kernel image
 - ▶ Variant: the compressed archive can also be loaded separately by the bootloader.
- ▶ It is useful for two cases
 - ▶ Fast booting of very small root filesystems. As the filesystem is completely loaded at boot time, application startup is very fast.
 - ▶ As an intermediate step before switching to a real root filesystem, located on devices for which drivers not part of the kernel image are needed (storage drivers, filesystem drivers, network drivers). This is always used on the kernel of desktop/server distributions to keep the kernel image size reasonable.



rootfs in memory: initramfs (2)

Kernel code and data

Root filesystem stored
as a compressed cpio
archive

Kernel image (zImage, bzImage, etc.)



rootfs in memory: initramfs (3)

- ▶ The contents of an initramfs are defined at the kernel configuration level, with the `CONFIG_INITRAMFS_SOURCE` option
 - ▶ Can be the path to a directory containing the root filesystem contents
 - ▶ Can be the path to a cpio archive
 - ▶ Can be a text file describing the contents of the initramfs (see documentation for details)
- ▶ The kernel build process will automatically take the contents of the `CONFIG_INITRAMFS_SOURCE` option and integrate the root filesystem into the kernel image
- ▶ Details (in kernel sources):
`Documentation/filesystems/ramfs-rootfs-initramfs.txt`
`Documentation/early-userspace/README`



Contents



Filesystem organization on GNU/Linux

- ▶ On most Linux based distributions, the filesystem layout is defined by the Filesystem Hierarchy Standard
- ▶ The FHS defines the main directories and their contents
 - /bin Essential command binaries
 - /boot Bootloader files, i.e. kernel images and related stuff
 - /etc Host-specific system-wide configuration files.
- ▶ Android follows an orthogonal path, storing its files in folders not present in the FHS, or following it when it uses a defined folder



Filesystem organization on Android

- ▶ Instead, the two main directories used by Android are
 - /system Immutable directory coming from the original build. It contains native binaries and libraries, framework jar files, configuration files, standard apps, etc.
 - /data is where all the changing content of the system are put: apps, data added by the user, data generated by all the apps at runtime, etc.
- ▶ These two directories are usually mounted on separate partitions, from the root filesystem originating from a kernel RAM disk.
- ▶ Android also uses some usual suspects: /proc, /dev, /sys, /etc, /sbin, /mnt where they serve the same function they usually do



`./app` All the pre-installed apps

`./bin` Binaries installed on the system (toolbox, vold, surfaceflinger)

`./etc` Configuration files

`./fonts` Fonts installed on the system

`./framework` Jar files for the framework

`./lib` Shared objects for the system libraries

`./modules` Kernel modules

`./xbin` External binaries



Other directories

- ▶ Like we said earlier, Android most of the time either uses directories not in the FHS, or directories with the exact same purpose as in standard Linux distributions (`/dev`, `/proc`, `/sys`), therefore avoiding collisions.
- ▶ There are some collisions though, for `/etc` and `/sbin`, which are hopefully trimmed down
- ▶ This allows to have a full Linux distribution side by side with Android with only minor tweaks



android_filesystem_config.h

- ▶ Located in system/core/include/private/
- ▶ Contains the full filesystem setup, and is written as a C header
 - ▶ UID/GID
 - ▶ Permissions for system directories
 - ▶ Permissions for system files
- ▶ Processed at compilation time to enforce the permissions throughout the filesystem
- ▶ Useful in other parts of the framework as well, such as ADB



Device Files



Devices

- ▶ One of the kernel important roles is to **allow applications to access hardware devices**
- ▶ In the Linux kernel, most devices are presented to user space applications through two different abstractions
 - ▶ **Character** device
 - ▶ **Block** device
- ▶ Internally, the kernel identifies each device by a triplet of information
 - ▶ **Type** (character or block)
 - ▶ **Major** (typically the category of device)
 - ▶ **Minor** (typically the identifier of the device)



Types of devices

- ▶ Block devices
 - ▶ A device composed of fixed-sized blocks, that can be read and written to store data
 - ▶ Used for hard disks, USB keys, SD cards, etc.
- ▶ Character devices
 - ▶ Originally, an infinite stream of bytes, with no beginning, no end, no size. The pure example: a serial port.
 - ▶ Used for serial ports, terminals, but also sound cards, video acquisition devices, frame buffers
 - ▶ Most of the devices that are not block devices are represented as character devices by the Linux kernel



Minimal filesystem

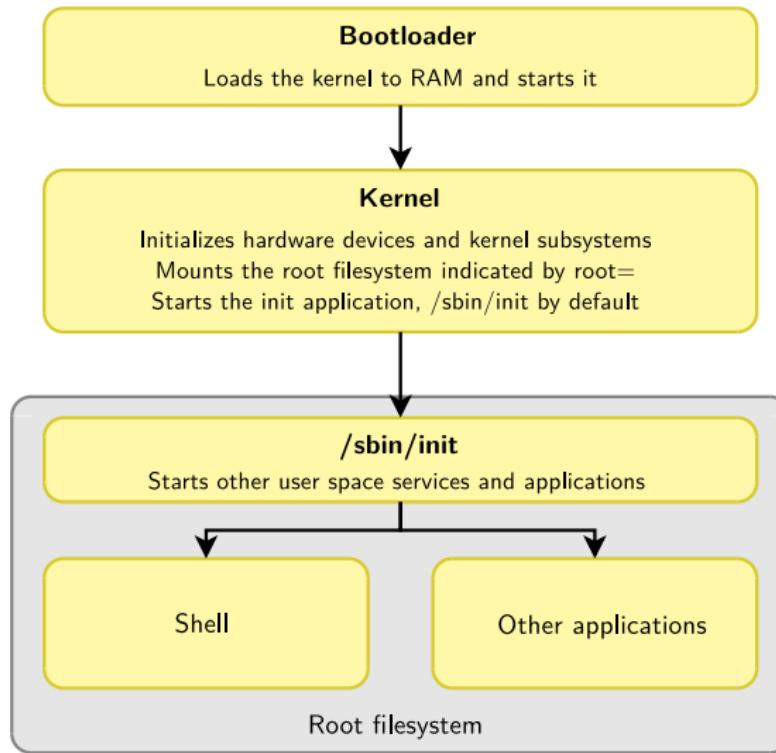


Basic applications

- ▶ In order to work, a Linux system needs at least a few applications
- ▶ An `init` application, which is the first user space application started by the kernel after mounting the root filesystem
 - ▶ The kernel tries to run `/sbin/init`, `/bin/init`, `/etc/init` and `/bin/sh`.
 - ▶ In the case of an initramfs, it will only look for `/init`. Another path can be supplied by the `rdinit` kernel argument.
 - ▶ If none of them are found, the kernel panics and the boot process is stopped.
 - ▶ The `init` application is responsible for starting all other user space applications and services
- ▶ A shell, to implement scripts, automate tasks, and allow a user to interact with the system
- ▶ Basic Unix executables, for use in system scripts or in interactive shells: `mv`, `cp`, `mkdir`, `cat`, `modprobe`, `mount`, `ifconfig`, etc.
- ▶ These basic components have to be integrated into the root filesystem to make it usable



Overall booting process

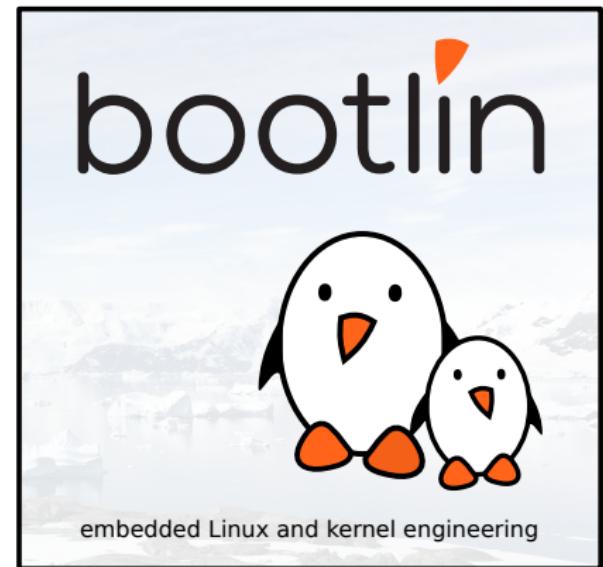




Android Build System: Advanced

Android Build System: Advanced

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Add a New Module



Modules

- ▶ Every component in Android is called a *module*
- ▶ Modules are defined across the entire tree through the `Android.mk` files
- ▶ The build system abstracts many details to make the creation of a module's Makefile as trivial as possible
- ▶ Of course, building a module that will be an Android application and building a static library will not require the same instructions, but these builds don't differ that much either.



Hello World

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES = hello_world.c
LOCAL_MODULE = HelloWorld

LOCAL_MODULE_TAGS = optional

include $(BUILD_EXECUTABLE)
```



Hello World

- ▶ Every module variable is prefixed by `LOCAL_*`
- ▶ `LOCAL_PATH` tells the build system where the current module is
- ▶ `include $(CLEAR_VARS)` cleans the previously declared `LOCAL_*` variables. This way, we make sure we won't have anything weird coming from other modules. The list of the variables cleared is in `build/core/clear_vars.mk`
- ▶ `LOCAL_SRC_FILES` contains a list of all source files to be compiled
- ▶ `LOCAL_MODULE` sets the module name
- ▶ `LOCAL_MODULE_TAGS` defines the set of modules this module should belong to
- ▶ `include $(BUILD_EXECUTABLE)` tells the build system to build this module as a binary



Tags

- ▶ Tags are used to define several sets of modules to be built through the build variant selected by `lunch`
- ▶ We have 3 build variants:
 - ▶ `user`
 - ▶ Installs modules tagged with `user`
 - ▶ Installs non-packaged modules that have no tags specified
 - ▶ `ro.secure = 1`
 - ▶ `ro.debuggable = 0`
 - ▶ ADB is disabled by default
 - ▶ `userdebug` is `user` plus
 - ▶ Installs modules tagged with `debug`
 - ▶ `ro.debuggable = 1`
 - ▶ ADB is enabled by default
 - ▶ `eng` is `userdebug`, plus
 - ▶ Installs modules tagged as `eng` and `development`
 - ▶ `ro.secure = 0`
 - ▶ `ro.kernel.android.checkjni = 1`
- ▶ Finally, we have a fourth tag, `optional`, that will never be directly integrated by a build variant, but deprecates `user`



Build Targets 1/3

- ▶ `BUILD_EXECUTABLE`
 - ▶ Builds a normal ELF binary to be run on the target
- ▶ `BUILD_HOST_EXECUTABLE`
 - ▶ Builds an ELF binary to be run on the host
- ▶ `BUILD_RAW_EXECUTABLE`
 - ▶ Builds a binary to be run on bare metal
- ▶ `BUILD_JAVA_LIBRARY`
 - ▶ Builds a Java library (`.jar`) to be used on the target
- ▶ `BUILD_STATIC_JAVA_LIBRARY`
 - ▶ Builds a static Java library to be used on the target



Build Targets 2/3

- ▶ `BUILD_HOST_JAVA_LIBRARY`
 - ▶ Builds a Java library to be used on the host
- ▶ `BUILD_SHARED_LIBRARY`
 - ▶ Builds a shared library for the target
- ▶ `BUILD_STATIC_LIBRARY`
 - ▶ Builds a static library for the target
- ▶ `BUILD_HOST_SHARED_LIBRARY`
 - ▶ Builds a shared library for the host
- ▶ `BUILD_HOST_STATIC_LIBRARY`
 - ▶ Builds a static library for the host
- ▶ `BUILD_RAW_STATIC_LIBRARY`
 - ▶ Builds a static library to be used on bare metal



Build Targets 3/3

- ▶ `BUILD_PREBUILT`
 - ▶ Used to install prebuilt files on the target (configuration files, kernel)
- ▶ `BUILD_HOST_PREBUILT`
 - ▶ Used to install prebuilt files on the host
- ▶ `BUILD_MULTI_PREBUILT`
 - ▶ Used to install prebuilt files of multiple modules of known types
- ▶ `BUILD_PACKAGE`
 - ▶ Builds a standard Android package (`.apk`)
- ▶ `BUILD_KEY_CHAR_MAP`
 - ▶ Builds a device character map



Other useful variables

- ▶ LOCAL_CFLAGS
 - ▶ Extra C compiler flags to use to build the module
- ▶ LOCAL_SHARED_LIBRARIES
 - ▶ List of shared libraries this module depends on at compilation time
- ▶ LOCAL_PACKAGE_NAME
 - ▶ Equivalent to LOCAL_MODULE for Android packages
- ▶ LOCAL_C_INCLUDES
 - ▶ List of paths to extra headers used by this module
- ▶ LOCAL_REQUIRED_MODULES
 - ▶ Express that a given module depends on another at runtime, and therefore should be included in the image as well
 - ▶ Many other similar options depending on what you want to do
 - ▶ You can get a complete list by reading
`build/core/clear_vars.mk`



Useful Make Macros

- ▶ In the build/core/definitions.mk file, you will find useful macros to use in the Android.mk file, that mostly allows you to:
 - ▶ Find files
 - ▶ all-makefiles-under, all-subdir-c-files, etc
 - ▶ Transform them
 - ▶ transform-c-to-o, ...
 - ▶ Copy them
 - ▶ copy-file-to-target, ...
 - ▶ and some utilities
 - ▶ my-dir, inherit-package, etc
- ▶ All these macros should be called through Make's call command, possibly with arguments



Prebuilt Package Example

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := configuration_files.txt
LOCAL_MODULE_CLASS := ETC
LOCAL_MODULE_PATH := $(TARGET_OUT_ETC)
LOCAL_SRC_FILES := $(LOCAL_MODULE)
include $(BUILD_PREBUILT)
```



Making and cleaning a module (1/2)

- ▶ To build a module from the top directory, just do
`make ModuleName`
- ▶ The files generated will be put in
`out/target/product/$TARGET_DEVICE/obj/<module_type>/<module_name>_intermediates`
- ▶ However, building a simple module won't regenerate a new image. This is just useful to make sure that the module builds. You will have to do a full `make` to have an image that contains your module
- ▶ Actually, a full `make` will build your module at some point, but you won't find it in your generated image if it is tagged as optional
- ▶ If you want to enable it for all builds, add its name to the `PRODUCT_PACKAGES` variables in the `build/target/product/core.mk` file.



Making and cleaning a module (2/2)

- ▶ To clean a single module, do `make clean-ModuleName`
- ▶ You can also get the list of the modules available in the build system with the `make modules` target



Practical lab - Building a Library



- ▶ Add an external library to the Android build system
- ▶ Compile it statically and dynamically
- ▶ Add a component to a build



Practical lab - Add a Native Application to the Build



- ▶ Add an external binary to a system
- ▶ Express dependencies on other components of the build system



Add a New Product

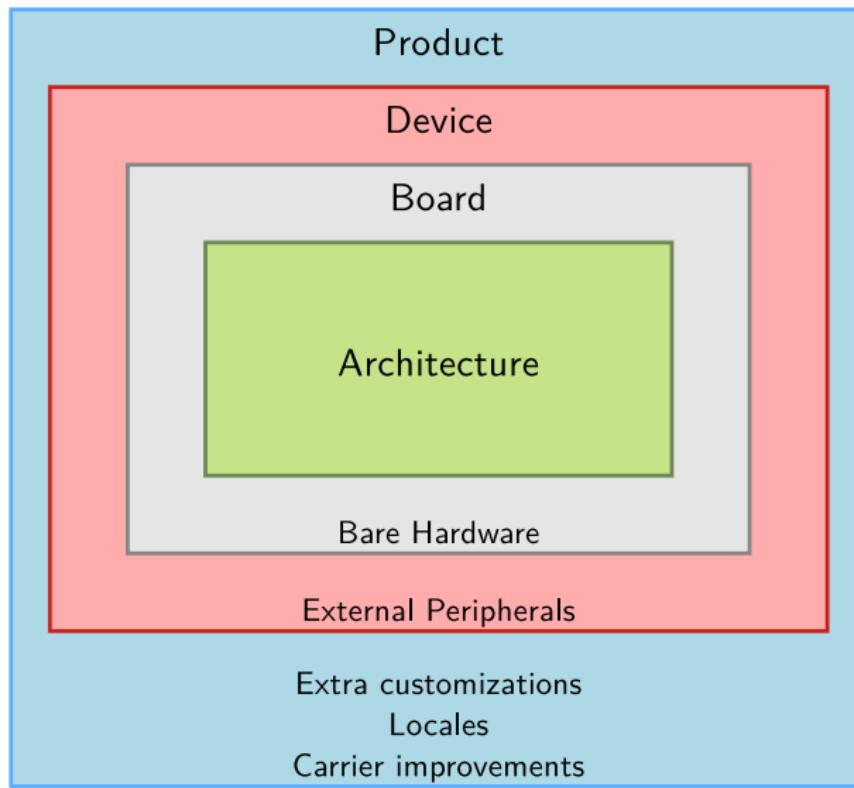


Defining new products

- ▶ Devices are well supported by the Android build system. It allows to build multiple devices with the same source tree, to have a per-device configuration, etc.
- ▶ All the product definitions should be put in `device/<company>/<device>`
- ▶ The entry point is the `AndroidProducts.mk` file, which should define the `PRODUCT_MAKEFILES` variable
- ▶ This variable defines where the actual product definitions are located.
- ▶ It follows such an architecture because you can have several products using the same device
- ▶ If you want your product to appear in the `lunch` menu, you need to create a `vendorsetup.sh` file in the `device` directory, with the right calls to `add_lunch_combo`



Product, devices and boards





Minimal Product Declaration

```
$(call inherit-product, build/target/product/generic.mk)
```

```
PRODUCT_NAME := full_MyDevice
```

```
PRODUCT_DEVICE := MyDevice
```

```
PRODUCT_MODEL := Full flavor of My Brand New Device
```



Copy files to the target

```
$(call inherit-product, build/target/product/generic.mk)

PRODUCT_COPY_FILES += \
    device/mybrand/mydevice/vold.fstab:system/etc/vold.fstab

PRODUCT_NAME := full_MyDevice
PRODUCT_DEVICE := MyDevice
PRODUCT_MODEL := Full flavor of My Brand New Device
```



Add a package to the build for this product

```
$(call inherit-product, build/target/product/generic.mk)

PRODUCT_PACKAGES += FooBar

PRODUCT_COPY_FILES += \
    device/mybrand/mydevice/vold.fstab:system/etc/vold.fstab

PRODUCT_NAME := full_mydevice
PRODUCT_DEVICE := mydevice
PRODUCT_MODEL := Full flavor of My Brand New Device
```



Overlays

- ▶ This is a mechanism used by products to override resources already defined in the source tree, without modifying the original code
- ▶ This is used for example to change the wallpaper for one particular device
- ▶ Use the `DEVICE_PACKAGE_OVERLAYS` or `PRODUCT_PACKAGE_OVERLAYS` variables that you set to a path to a directory in your device folder
- ▶ This directory should contain a structure similar to the source tree one, with only the files that you want to override



Add a device overlay

```
$(call inherit-product, build/target/product/generic.mk)

PRODUCT_PACKAGES += FooBar

PRODUCT_COPY_FILES += \
    device/mybrand/mydevice/vold.fstab:system/etc/vold.fstab

DEVICE_PACKAGE_OVERLAYS := device/mybrand/mydevice/overlay

PRODUCT_NAME := full_mydevice
PRODUCT_DEVICE := mydevice
PRODUCT_MODEL := Full flavor of My Brand New Device
```



Board Definition

- ▶ You will also need a `BoardConfig.mk` file along with the product definition
- ▶ While the product definition was mostly about the build system in itself, the board definition is more about the hardware
- ▶ However, this is poorly documented and sometimes ambiguous so you will probably have to dig into the `build/core/Makefile` at some point to see what a given variable does



Minimal Board Definition

```
TARGET_NO_BOOTLOADER := true  
TARGET_NO_KERNEL := true  
TARGET_CPU_ABI := armeabi  
HAVE_HTC_AUDIO_DRIVER := true  
BOARD_USES_GENERIC_AUDIO := true  
  
USE_CAMERA_STUB := true
```



Other Board Variables 1/2

- ▶ TARGET_ARCH_VARIANT
 - ▶ Variant of the selected architecture (for example armv7-a-neon for most Cortex-A8 and A9 CPUs)
- ▶ TARGET_EXTRA_CFLAGS
 - ▶ Extra C compiler flags to use during the whole build
- ▶ TARGET_CPU_SMP
 - ▶ Does the CPU have multiple cores?
- ▶ TARGET_USERIMAGES_USE_EXT4
 - ▶ We want to use ext4 filesystems for our generated partitions
- ▶ BOARD_SYSTEMIMAGE_PARTITION_SIZE
 - ▶ Size of the system partitions in bytes.



Other Board Variables 2/2

- ▶ BOARD_NAND_PAGE_SIZE
 - ▶ For NAND flash, size of the pages as given by the datasheet
- ▶ TARGET_NO_RECOVERY
 - ▶ We don't want to build the recovery image
- ▶ BOARD_KERNEL_CMDLINE
 - ▶ Boot arguments of the kernel



Kernel Integration into Android

- ▶ Android is just a user space software stack, the build system isn't designed to build the kernel
- ▶ However, there is some facilities to integrate a precompiled kernel into an Android image
- ▶ To do so, you need to:
 - ▶ In `BoardConfig.mk`
 - ▶ Remove `TARGET_NO_KERNEL` if set
 - ▶ Set `BOARD_KERNEL_BASE` to the load address of your kernel
 - ▶ In your device Makefile, have something like

```
ifeq ($(TARGET_PREBUILT_KERNEL),)  
LOCAL_KERNEL := device/ti/panda/kernel  
else  
LOCAL_KERNEL := $(TARGET_PREBUILT_KERNEL)  
endif
```



```
PRODUCT_COPY_FILES := \  
$(LOCAL_KERNEL):kernel
```



Practical lab - System Customization



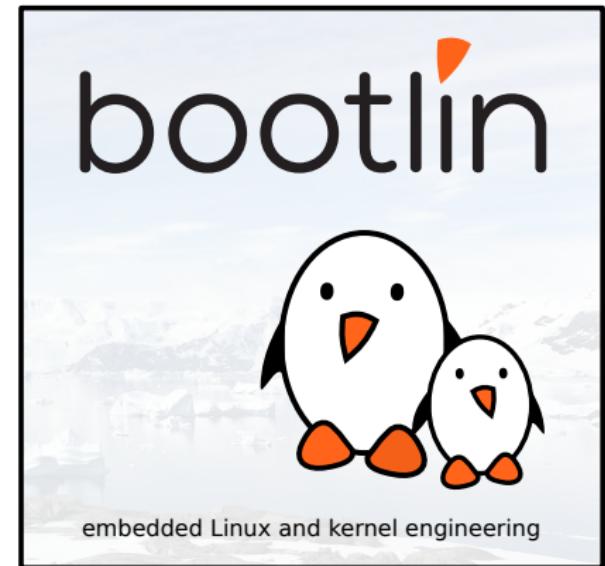
- ▶ Use the product configuration system
- ▶ Change the default wallpaper
- ▶ Add extra properties to the system
- ▶ Use the product overlays



Android Native Layer

Android Native Layer

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Definition and Components

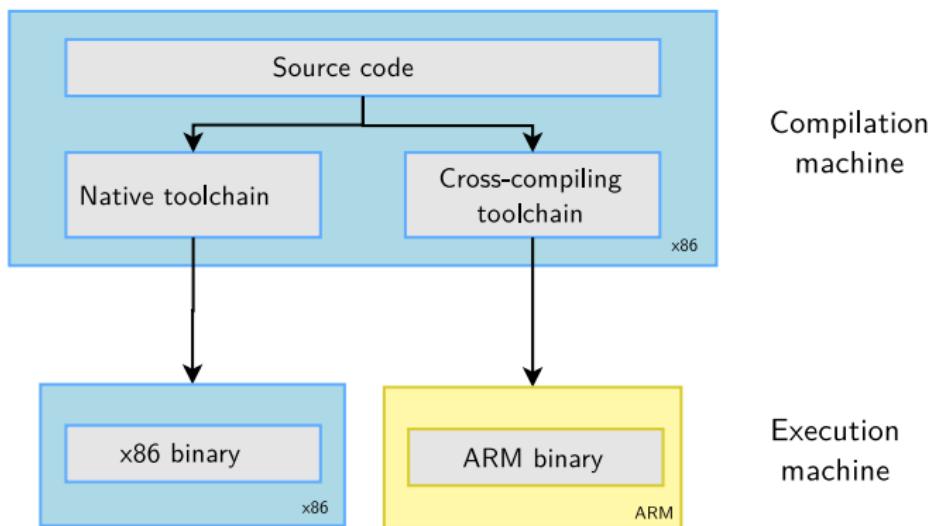


Definition (1)

- ▶ The usual development tools available on a GNU/Linux workstation is a **native toolchain**
- ▶ This toolchain runs on your workstation and generates code for your workstation, usually x86
- ▶ For embedded system development, it is usually impossible or not interesting to use a native toolchain
 - ▶ The target is too restricted in terms of storage and/or memory
 - ▶ The target is very slow compared to your workstation
 - ▶ You may not want to install all development tools on your target.
- ▶ Therefore, **cross-compiling toolchains** are generally used. They run on your workstation but generate code for your target.



Definition (2)





Machines in build procedures

- ▶ Three machines must be distinguished when discussing toolchain creation
 - ▶ The **build** machine, where the toolchain is built.
 - ▶ The **host** machine, where the toolchain will be executed.
 - ▶ The **target** machine, where the binaries created by the toolchain are executed.
- ▶ Four common build types are possible for toolchains



Different toolchain build procedures



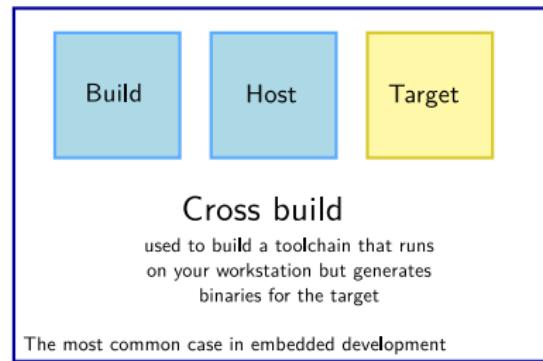
Native build

used to build the normal gcc
of a workstation



Cross-native build

used to build a toolchain that runs on your
target and generates binaries for the target



Cross build

used to build a toolchain that runs on your
workstation but generates
binaries for the target

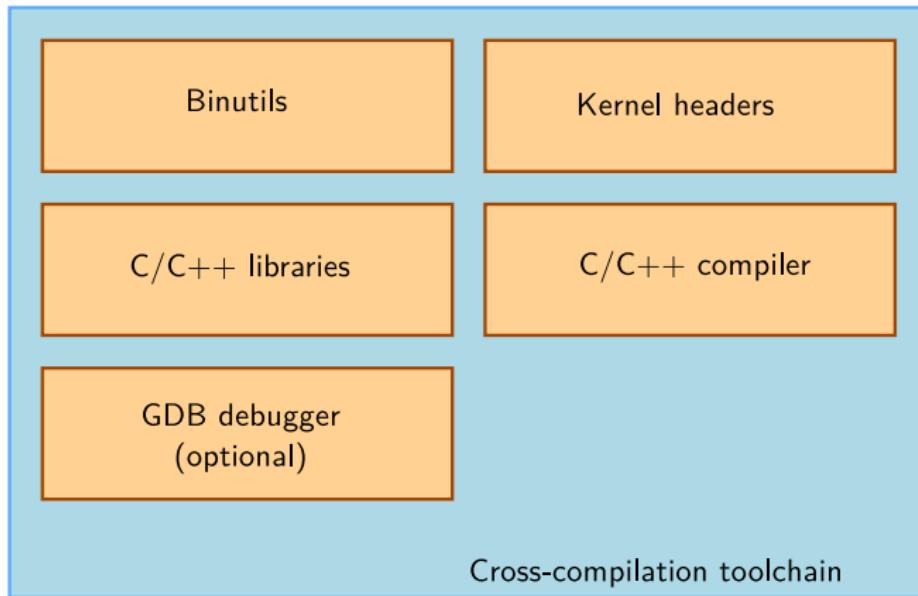


Canadian cross build

used to build on architecture A
a toolchain that runs on architecture B
and generates binaries for architecture C



Components



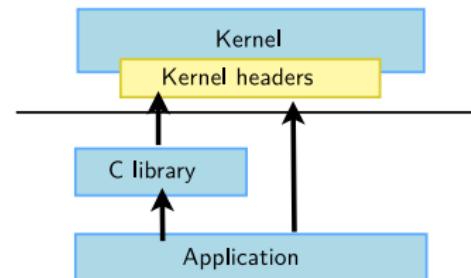


- ▶ **Binutils** is a set of tools to generate and manipulate binaries for a given CPU architecture
 - ▶ as, the assembler, that generates binary code from assembler source code
 - ▶ ld, the linker
 - ▶ ar, ranlib, to generate .a archives, used for libraries
 - ▶ objdump, readelf, size, nm, strings, to inspect binaries.
Very useful analysis tools!
 - ▶ objcopy, to modify binaries
 - ▶ strip, to strip parts of binaries that are just needed for debugging (reducing their size).
- ▶ <http://www.gnu.org/software/binutils/>
- ▶ GPL license



Kernel headers (1)

- ▶ The C library and compiled programs needs to interact with the kernel
 - ▶ Available system calls and their numbers
 - ▶ Constant definitions
 - ▶ Data structures, etc.
- ▶ Therefore, compiling the C library requires kernel headers, and many applications also require them.
- ▶ Available in `<linux/...>` and `<asm/...>` and a few other directories corresponding to the ones visible in `include/` in the kernel sources





Kernel headers (2)

- ▶ System call numbers, in `<asm/unistd.h>`

```
#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
```

- ▶ Constant definitions, here in `<asm-generic/fcntl.h>`, included from `<asm/fcntl.h>`, included from `<linux/fcntl.h>`

```
#define O_RDWR 00000002
```

- ▶ Data structures, here in `<asm/stat.h>`

```
struct stat {
    unsigned long st_dev;
    unsigned long st_ino;
    [...]
};
```



Kernel headers (3)

- ▶ The kernel to user space ABI is **backward compatible**
 - ▶ Binaries generated with a toolchain using kernel headers older than the running kernel will work without problem, but won't be able to use the new system calls, data structures, etc.
 - ▶ Binaries generated with a toolchain using kernel headers newer than the running kernel might work on if they don't use the recent features, otherwise they will break
 - ▶ Using the latest kernel headers is not necessary, unless access to the new kernel features is needed
- ▶ The kernel headers are extracted from the kernel sources using the `headers_install` kernel Makefile target.



C/C++ compiler

- ▶ GCC: GNU Compiler Collection, the famous free software compiler
- ▶ <http://gcc.gnu.org/>
- ▶ Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, Go, etc. Can generate code for a large number of CPU architectures, including ARM, AVR, Blackfin, CRIS, FRV, M32, MIPS, MN10300, PowerPC, SH, v850, x86, x86_64, IA64, Xtensa, etc.
- ▶ Available under the GPL license, libraries under the GPL with linking exception.
- ▶ Alternative: Clang / LLVM compiler (<http://clang.llvm.org/>) getting increasingly popular and able to compile most programs (license: MIT/BSD type)



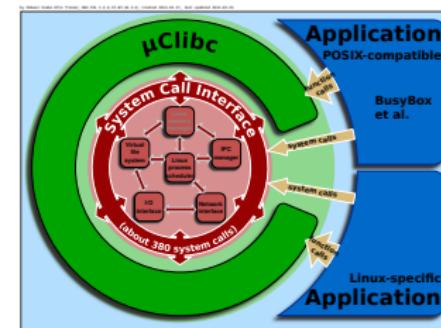


C library

- ▶ The C library is an essential component of a Linux system
 - ▶ Interface between the applications and the kernel
 - ▶ Provides the well-known standard C API to ease application development
- ▶ Several C libraries are available: *glibc*, *uClibc*, *musl*, *klibc*, *newlib*...
- ▶ The choice of the C library must be made at cross-compiling toolchain generation time, as the GCC compiler is compiled against a specific C library.

Comparing libcs by feature:

http://www.etalabs.net/compare_libcs.html



Source: Wikipedia (<http://bit.ly/2zrGve2>)

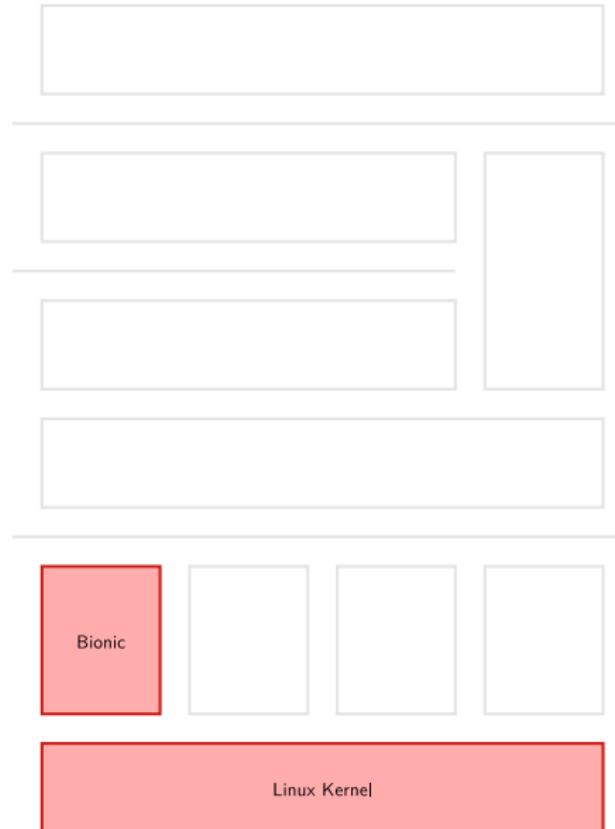


Android Native Layer

Bionic



Whole Android Stack





- ▶ Google developed another C library for Android: Bionic.
They didn't start from scratch however, they based their work on the BSD standard C library.
- ▶ The most remarkable thing about Bionic is that it doesn't have full support for the POSIX API, so it might be a hurdle when porting an already developed program to Android.
- ▶ Among other things, are lacking:
 - ▶ Full pthreads API
 - ▶ No locales and wide chars support
 - ▶ No `openpty()`, `syslog()`, `crypt()`, functions
 - ▶ Removed dependency on the `/etc/resolv.conf` and `/etc/passwd` files and using Android's own mechanisms instead
 - ▶ Some functions are still unimplemented (see `getprotobynumber()`)



Bionic 2/2

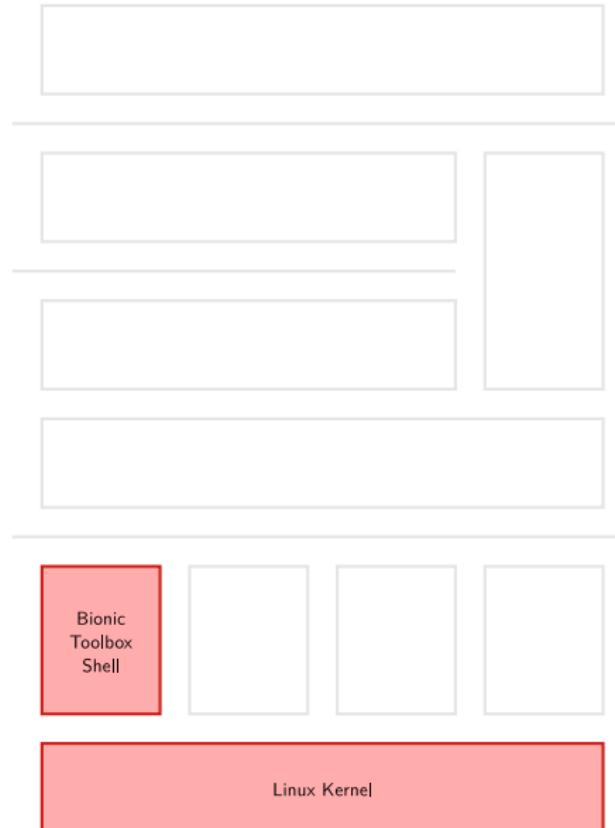
- ▶ However, Bionic has been created this way for a number of reasons
 - ▶ Keep the libc implementation as simple as possible, so that it can be fast and lightweight (Bionic is a bit smaller than uClibc)
 - ▶ Keep the (L)GPL code out of the user space. Bionic is under the BSD license
- ▶ And it implements some Android-specifics functions as well:
 - ▶ Access to system properties
 - ▶ Logging events in the system logs
- ▶ In the `prebuilt/` directory, Google provides a prebuilt toolchain that uses Bionic
- ▶ See <http://androidxref.com/4.0.4/xref/ndk/docs/system/libc/OVERVIEW.html> for details about Bionic.



Toolbox



Whole Android Stack





Why Toolbox?

- ▶ A Linux system needs a basic set of programs to work
 - ▶ An init program
 - ▶ A shell
 - ▶ Various basic utilities for file manipulation and system configuration
- ▶ In normal Linux systems, these programs are provided by different projects
 - ▶ coreutils, bash, grep, sed, tar, wget, modutils, etc. are all different projects
 - ▶ Many different components to integrate
 - ▶ Components not designed with embedded systems constraints in mind: they are not very configurable and have a wide range of features
- ▶ Busybox is an alternative solution, extremely common on embedded systems



General purpose toolbox: BusyBox

- ▶ Rewrite of many useful Unix command line utilities
 - ▶ Integrated into a single project, which makes it easy to work with
 - ▶ Designed with embedded systems in mind: highly configurable, no unnecessary features
- ▶ All the utilities are compiled into a single executable, /bin/busybox
 - ▶ Symbolic links to /bin/busybox are created for each application integrated into Busybox
- ▶ For a fairly featureful configuration, less than 500 KB (statically compiled with uClibc) or less than 1 MB (statically compiled with glibc).
- ▶ <http://www.busybox.net/>



BusyBox commands!

Commands available in BusyBox 1.13

[, [[, addgroup, adduser, adjtimex, ar, arp, arping, ash, awk, basename, bbconfig, bbsht, 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, mkttemp, modprobe, more, mount, mountpoint, msh, mt, mv, nameif, nc, 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, resize, restorecon, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run_parts, runcon, runlevel, runsv, runsvdir, rx, script, sed, selinuxenabled, sendmail, seq, sestatus, setarch, setconsole, setenforce, setfiles, setfont, setkeycodes, setlogcons, setsebool, setsid, setuidgid, sh, sha1sum, showkey, slattach, sleep, softlimit, sort, split, start_stop_daemon, stat, strings, stty, su, slogin, 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, uniq, unix2dos, unlzma, unzip, uptime, usleep, uudecode, uuencode, vconfig, vi, vlock, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat, zcip



- ▶ As Busybox is under the GPL, Google developed an equivalent tool, under the BSD license
- ▶ Much fewer UNIX commands implemented than Busybox, but other commands to use the Android-specifics mechanism, such as alarm, getprop or a modified log

Commands available in Toolbox in Jelly Bean

alarm, cat, chcon, chmod, chown, cmp, cp, date, dd, df, dmesg, du, dynarray, exists, getenforce, getevent, getprop, getsebool, grep, hd, id, ifconfig, iftop, insmod, ioctl, ionice, kill, ln, load_policy, log, ls, lsmod, lsof, lsusb, md5, mkdir, mount, mv, nandread, netstat, newfs_msdos, notify, printenv, ps, r, readyt, reboot, renice, restorecon, rm, rmdir, rmmod, rotatefb, route, runcon, schedtop, sendevent, setconsole, setenforce, setkey, setprop, setsebool, sleep, smd, start, stop, sync, syren, top, touch, umount, uptime, vmstat, watchprops, wipe

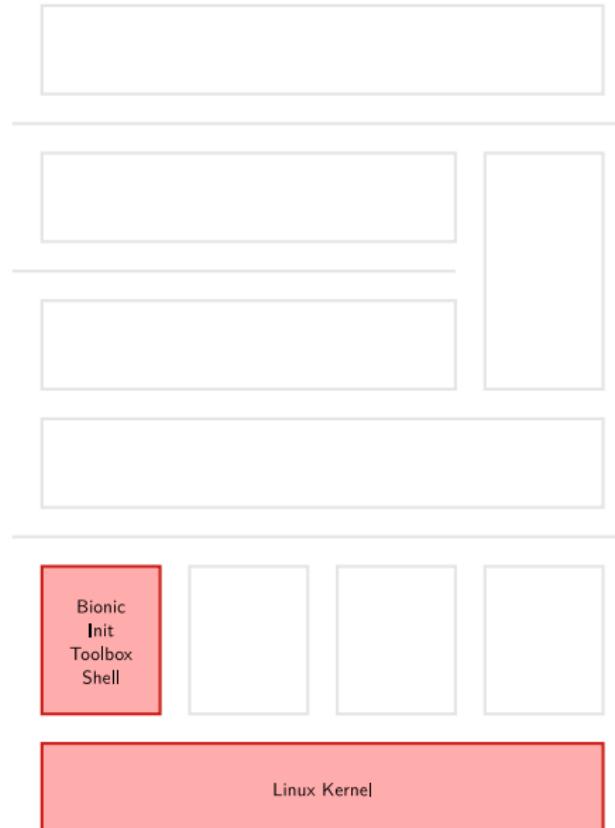
- ▶ The shell is provided by an external project, mksh, which is a BSD-licenced implementation of ksh



Init



Whole Android Stack





Init

- ▶ init is the name of the first user space program
- ▶ It is up to the kernel to start it, with PID 1, and the program should never exit during system life
- ▶ The kernel will look for init at `/sbin/init`, `/bin/init`, `/etc/init` and `/bin/sh`. You can tweak that with the `init=kernel` parameter
- ▶ The role of init is usually to start other applications at boot time, a shell, mount the various filesystems, etc.
- ▶ Init also manages the shutdown of the system by undoing all it has done at boot time



Android's init

- ▶ Once again, Google has developed his own instead of relying on an existing one.
- ▶ However, it has some interesting features, as it can also be seen as a daemon on the system
 - ▶ it manages device hotplugging, with basic permissions rules for device files, and actions at device plugging and unplugging
 - ▶ it monitors the services it started, so that if they crash, it can restart them
 - ▶ it monitors system properties so that you can take actions when a particular one is modified



Init part

- ▶ For the initialization part, init mounts the various filesystems (/proc, /sys, data, etc.)
- ▶ This allows to have an already setup environment before taking further actions
- ▶ Once this is done, it reads the `init.rc` file and executes it



init.rc file interpretation

- ▶ Uses a unique syntax, based on events
- ▶ There usually are several init configuration files, the main `init.rc` file itself, plus the extra file included from it
- ▶ By default, these included files hold either subsystem-specific initialisation (USB, Kernel Tracing), or hardware-specific instructions
- ▶ It relies on system properties, evaluated at runtime, that allows to have on the same system, configuration for several different platforms, that will be used only when they are relevant.
- ▶ Most of the customizations should therefore go to the platform-specific configuration file rather than to the generic one



Syntax

- ▶ Unlike most init script systems, the configuration relies on system event and system property changes, allowed by the daemon part of it
- ▶ This way, you can trigger actions not only at startup or at run-level changes like with traditional init systems, but also at a given time during system life



Actions

```
on <trigger>
    command
    command
```

- ▶ Here are a few trigger types:
 - ▶ boot
 - ▶ Triggered when init is loaded
 - ▶ <property>=<value>
 - ▶ Triggered when the given property is set to the given value
 - ▶ device-added-<path>
 - ▶ Triggered when the given device node is added or removed
 - ▶ service-exited-<name>
 - ▶ Triggered when the given service exits



Init triggers

- ▶ Commands are also specific to Android, with sometimes a syntax very close to the shell one (just minor differences):
- ▶ The complete list of triggers, by execution order is:
 - ▶ early-init
 - ▶ init
 - ▶ early-fs
 - ▶ fs
 - ▶ post-fs
 - ▶ early-boot
 - ▶ boot



Example

```
import /init.${ro.hardware}.rc

on boot
    export PATH /sbin:/system/sbin:/system/bin
    export LD_LIBRARY_PATH /system/lib

    mkdir /dev
    mkdir /proc
    mkdir /sys

    mount tmpfs tmpfs /dev
    mkdir /dev/pts
    mkdir /dev/socket
    mount devpts devpts /dev/pts
    mount proc proc /proc
    mount sysfs sysfs /sys
```



Services

```
service <name> <pathname> [ <argument> ]*  
    <option>  
    <option>
```

- ▶ Services are like daemons
- ▶ They are started by init, managed by it, and can be restarted when they exit
- ▶ Many options, ranging from which user to run the service as, rebooting in recovery when the service crashes too frequently, to launching a command at service reboot.



Example

```
on device-added-/dev/compass  
    start akmd
```

```
on device-removed-/dev/compass  
    stop akmd
```

```
service akmd /sbin/akmd  
    disabled  
    user akmd  
    group akmd
```



Uevent

- ▶ Init also manages the runtime events generated by the kernel when hardware is plugged in or removed, like udev does on a standard Linux distribution
- ▶ This way, it dynamically creates the devices nodes under `/dev`
- ▶ You can also tweak its behavior to add specific permissions to the files associated to a new event.
- ▶ The associated configuration files are `/ueventd.rc` and `/ueventd.<platform>.rc`
- ▶ While `ueventd.rc` is always taken into account, `ueventd.<platform>.rc` is only interpreted if the platform currently running the system reports the same name
- ▶ This name is either obtained by reading the file `/proc/cpuinfo` or from the `androidboot.hardware` kernel parameter



ueventd.rc syntax

<path> <permission> <user> <group>

► Example

/dev/bus/usb/* 0660 root usb



Properties

- ▶ Init also manages the system properties
- ▶ Properties are a way used by Android to share values across the system that are not changing quite often
- ▶ Quite similar to the Windows Registry
- ▶ These properties are splitted into several files:
 - ▶ `/system/build.prop` which contains the properties generated by the build system, such as the date of compilation
 - ▶ `/default.prop` which contains the default values for certain key properties, mostly related to the security and permissions for ADB.
 - ▶ `/data/local.prop` which contains various properties specific to the device
 - ▶ `/data/property` is a folder which purpose is to be able to edit properties at run-time and still have them at the next reboot. This folder is storing every properties prefixed by `persist.` in separate files containing the values.



Modifying Properties

- ▶ You can add or modify properties in the build system by using either the `PRODUCT_PROPERTY_OVERRIDES` makefile variable, or by defining your own `system.prop` file in the device directory. Their content will be appended to `/system/build.prop` at compilation time
- ▶ Modify the `init.rc` file so that at boot time it exports these properties using the `setprop` command
- ▶ Using the API functions such as the Java function `SystemProperties.set`



Permissions on the Properties

- ▶ Android, by default, only allows any given process to read the properties.
- ▶ You can set write permissions on a particular property or a group of them using the file system/core/init/property_service.c

```
/* White list of permissions for setting property services. */
struct {
    const char *prefix;
    unsigned int uid;
    unsigned int gid;
} property_perms[] = {
    { "net.rmneth0.",      AID_RADIO,      0 },
    { "net.dns",           AID_RADIO,      0 },
    { "net.",              AID_SYSTEM,     0 },
    { "dhcp.",             AID_SYSTEM,     0 },
    { "log.",              AID_SHELL,      0 },
    { "service.adb.root",  AID_SHELL,      0 },
    { "persist.security.", AID_SYSTEM,     0 },
    { NULL, 0, 0 }
};
```



Special Properties

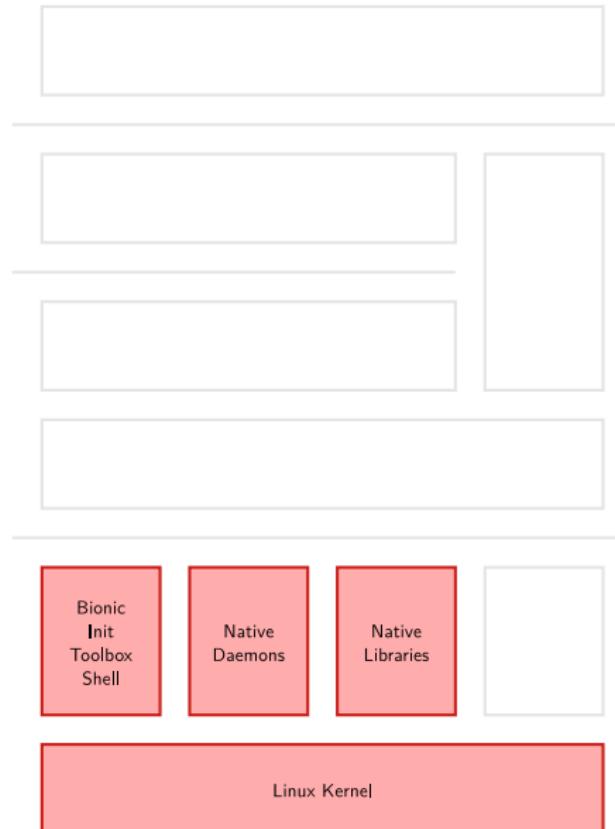
- ▶ `ro.*` properties are read-only. They can be set only once in the system life-time. You can only change their value by modifying the property files and reboot.
- ▶ `persist.*` properties are stored on persistent storage each time they are set.
- ▶ `ctl.start` and `ctl.stop` properties used instead of storing properties to start or stop the service name passed as the new value
- ▶ `net.change` property holds the name of the last `net.*` property changed.



Various daemons



Whole Android Stack





- ▶ The VOLUME Daemon
- ▶ Just like init does, monitors new device events
- ▶ While init was only creating device files and taking some configured options, vold actually only cares about storage devices
- ▶ Its roles are to:
 - ▶ Auto-mount the volumes
 - ▶ Format the partitions on the device
- ▶ There is no /etc/fstab in Android, but /system/etc/vold.fstab has a somewhat similar role



- ▶ rild is the Radio Interface Layer Daemon
- ▶ This daemon drives the telephony stack, both voice and data communication
- ▶ When using the voice mode, talks directly to the baseband, but when issuing data transfers, relies on the kernel network stack
- ▶ It can handle two types of commands:
 - ▶ *Solicited commands*: commands that originate from the user: dial a number, send an SMS, etc.
 - ▶ *Unsolicited commands*: commands that come from the baseband: receiving an SMS, a call, signal strength changed, etc.



Others

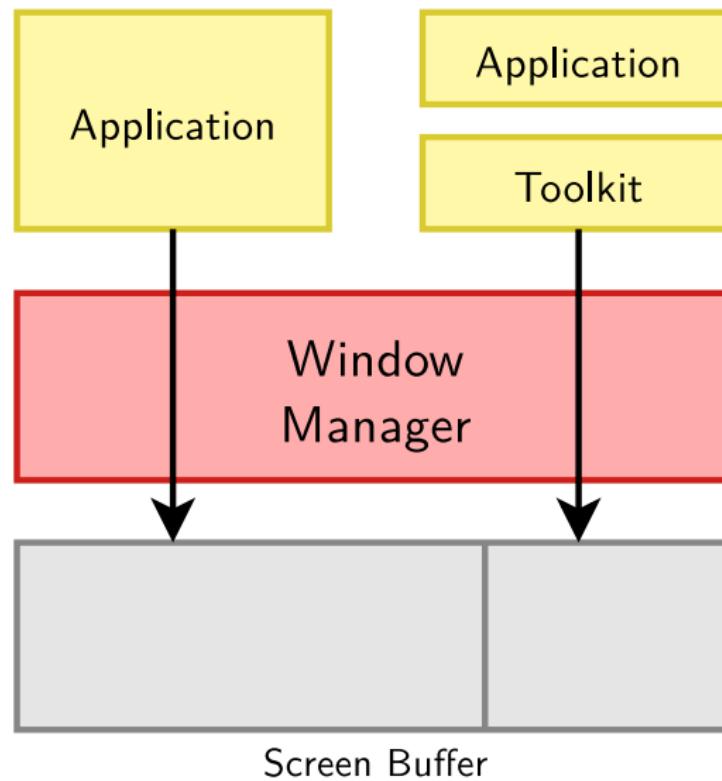
- ▶ netd
 - ▶ netd manages the various network connections: Bluetooth, Wifi, USB
 - ▶ Also takes any associated actions: detect new connections, set up the tethering, etc.
 - ▶ It really is an equivalent to NetworkManager
 - ▶ On a security perspective, it also allows to isolate network-related privileges in a single process
- ▶ installd
 - ▶ Handles package installation and removal
 - ▶ Also checks package integrity, installs the native libraries on the system, etc.



SurfaceFlinger

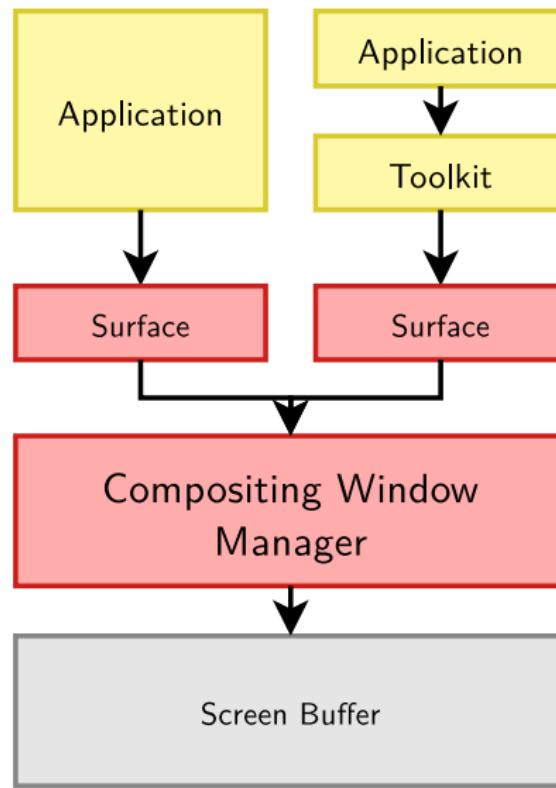


Introduction to graphical stacks





Compositing window managers

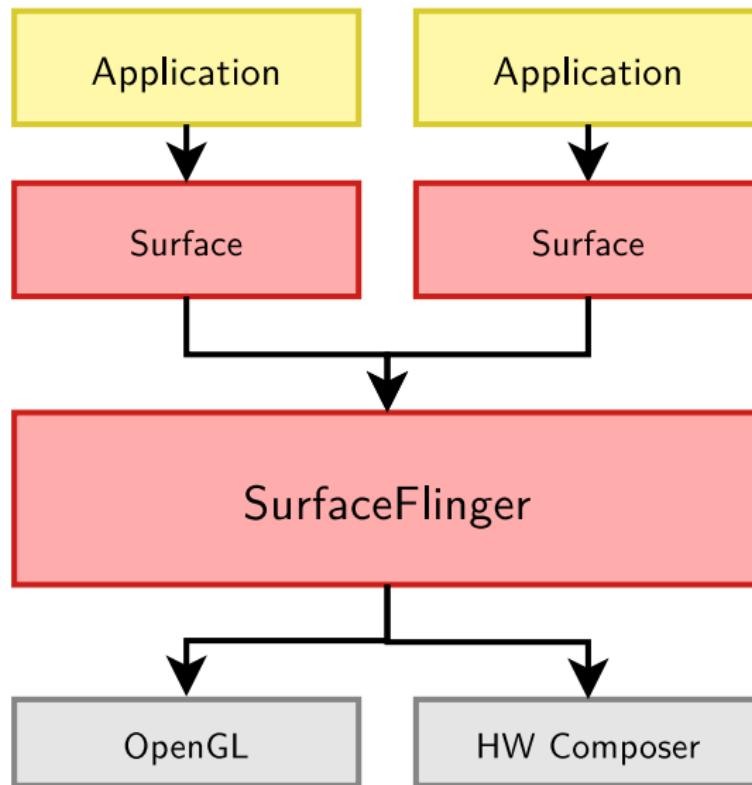




- ▶ This difference in design adds some interesting features:
 - ▶ Effects are easy to implement, as it's up to the window manager to mangle the various surfaces at will to display them on the screen. Thus, you can add transparency, 3d effects, etc.
 - ▶ Improved stability. With a regular window manager, a message is sent to every window to redraw its part of the screen, for example when a window has been moved. But if an application fails to redraw, the windows will become glitchy. This will not happen with a compositing WM, as it will still display the untouched surface.
- ▶ SurfaceFlinger is the compositing window manager in Android, providing surfaces to applications and rendering all of them with hardware acceleration.



SurfaceFlinger





Stagefright

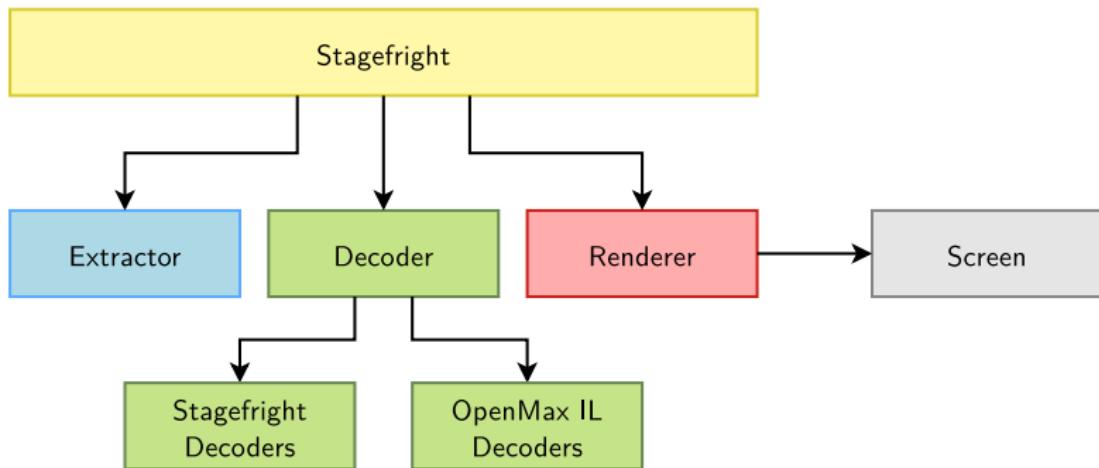


Stagefright

- ▶ StageFright is the multimedia playback engine in Android since Eclair
- ▶ In its goals, it is quite similar to GStreamer: Provide an abstraction on top of codecs and libraries to easily play multimedia files
- ▶ It uses a plugin system, to easily extend the number of formats supported, either software or hardware decoded



StageFright Architecture





StageFright plugins

- ▶ To add support for a new format, you need to:
 - ▶ Develop a new Extractor class, if the container is not supported yet.
 - ▶ Develop a new Decoder class, that implements the interface needed by the StageFright core to read the data.
 - ▶ Associate the mime-type of the files to read to your new Decoder in the `/etc/media_codecs.xml` file provided by your device, either in the Decoders list.
 - ▶ → No runtime extension of the decoders, this is done at compilation time.

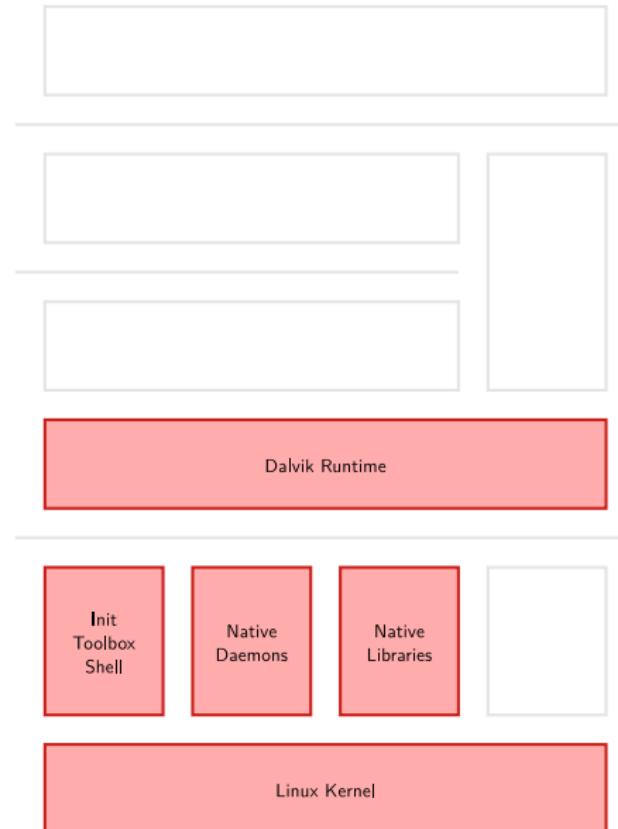
```
<Decoders>
    <MediaCodec name="OMX.google.vorbis.decoder" type="audio/vorbis" />
    <MediaCodec name="OMX.qcom.video.decoder.avc" type="video/avc" />
</Decoders>
```



Dalvik and Zygote



Whole Android Stack





- ▶ Dalvik is the virtual machine, executing Android applications
- ▶ It is an interpreter written in C/C++, and is designed to be portable, lightweight and run well on mobile devices
- ▶ It is also designed to allow several instances of it to be run at the same time while consuming as little memory as possible
- ▶ Two execution modes
 - ▶ portable: the interpreter is written in C, quite slow, but should work on all platforms
 - ▶ fast: Uses the *mterp* mechanism, to define routines either in assembly or in C optimized for a specific platform. Instruction dispatching is also done by computing the handler address from the opcode number
- ▶ It uses the *Apache Harmony* Java framework for its core libraries



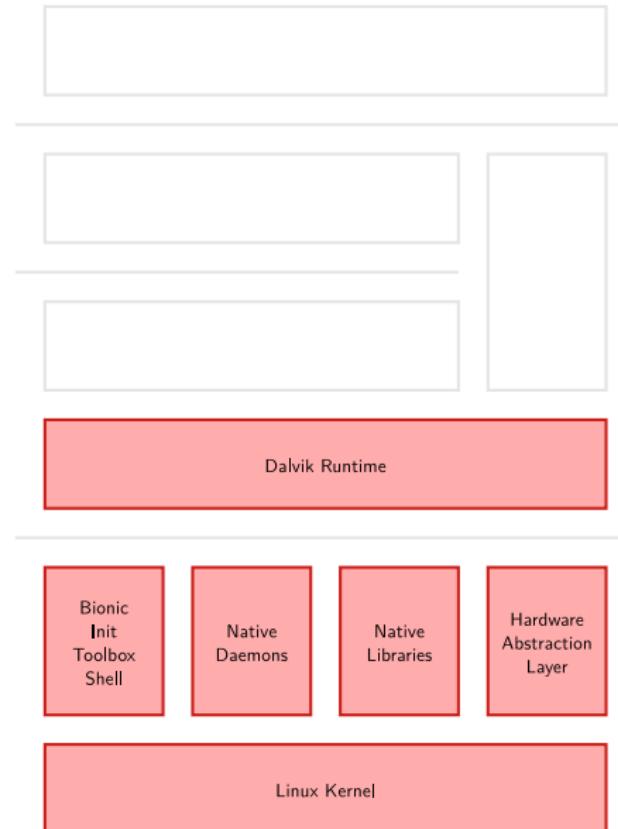
- ▶ Dalvik is started by Zygote
- ▶ frameworks/base/cmds/app_process
- ▶ At boot, Zygote is started by init, it then
 - ▶ Initializes a virtual machine in its address space
 - ▶ Loads all the basic Java classes in memory
 - ▶ Starts the system server
 - ▶ Waits for connections on a UNIX socket
- ▶ When a new application should be started:
 - ▶ Android connects to Zygote through the socket to request the start of a new application
 - ▶ Zygote forks
 - ▶ The child process loads the new application and start executing it



Hardware Abstraction Layer



Whole Android Stack





Hardware Abstraction Layers

- ▶ Usually, the kernel already provides a HAL for user space
- ▶ However, from Google's point of view, this HAL is not sufficient and suffers some restrictions, mostly:
 - ▶ Depending on the subsystem used in the kernel, the user space interface differs
 - ▶ All the code in the kernel must be GPL-licensed
- ▶ Google implemented its HAL with dynamically loaded user space libraries



Library naming

- ▶ It follows the same naming scheme as for init: the generic implementation is called `libfoo.so` and the hardware-specific one `libfoo.hardware.so`
- ▶ The name of the hardware is looked up with the following properties:
 - ▶ `ro.hardware`
 - ▶ `ro.product.board`
 - ▶ `ro.board.platform`
 - ▶ `ro.arch`
- ▶ The libraries are then searched for in the directories:
 - ▶ `/vendor/lib/hw`
 - ▶ `/system/lib/hw`



Various layers

- ▶ Audio (`libaudio.so`) configuration, mixing, noise cancellation, etc.
 - ▶ `hardware/libhardware/include/audio.h`
- ▶ Graphics (`gralloc.so`, `hwcomposer.so`, `libhgl.so`) handles graphic memory buffer allocations, OpenGL implementation, etc.
 - ▶ `libhgl.so` should be provided by your vendor
 - ▶ `hardware/libhardware/include/gralloc.h`
 - ▶ `hardware/libhardware/include/hwcomposer.h`
- ▶ Camera (`libcamera.so`) handles the camera functions: autofocus, take a picture, etc.
 - ▶ `hardware/libhardware/include/camera{2,3}.h`
- ▶ GPS (`libgps.so`) configuration, data acquisition
 - ▶ `hardware/libhardware/include/hardware/gps.h`

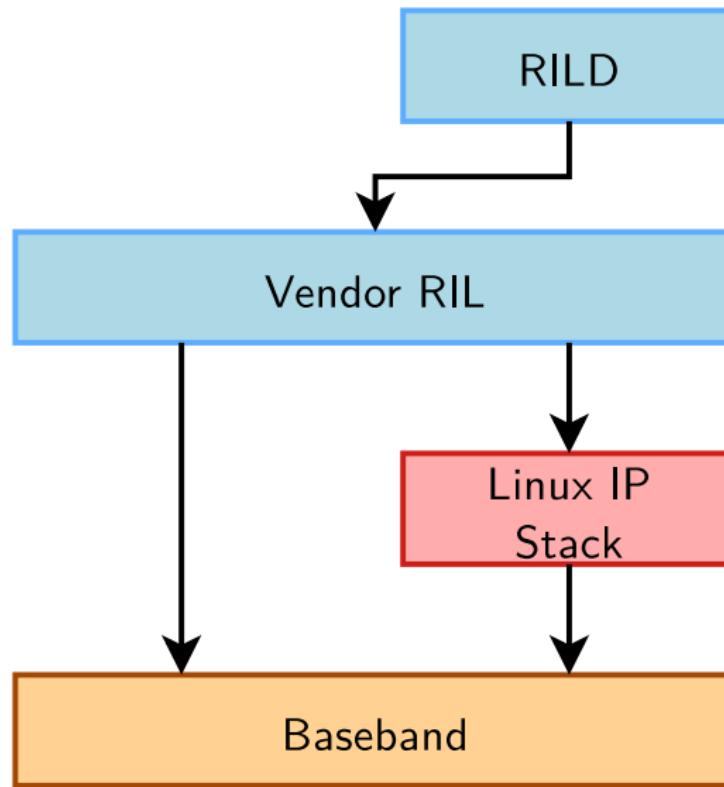


Various layers

- ▶ **Lights** (`liblights.so`) Backlight and LEDs management
 - ▶ `hardware/libhardware/include/lights.h`
- ▶ **Sensors** (`libsensors.so`) handles the various sensors on the device: Accelerometer, Proximity Sensor, etc.
 - ▶ `hardware/libhardware/include/sensors.h`
- ▶ **Radio Interface** (`libril-vendor-version.so`) manages all communication between the baseband and `rild`
 - ▶ You can set the name of the library with the `rild.lib` and `rild.libargs` properties to find the library
 - ▶ `hardware/ril/include/telephony/ril.h`
- ▶ **Bluetooth** (`libbluetooth.so`) Discovery and communication with Bluetooth devices
 - ▶ `hardware/libhardware/include/bluetooth.h`
- ▶ **NFC** (`libnfc.so`) Discover NFC devices, communicate with it, etc.
 - ▶ `hardware/libhardware/include/nfc.h`



Example: rild



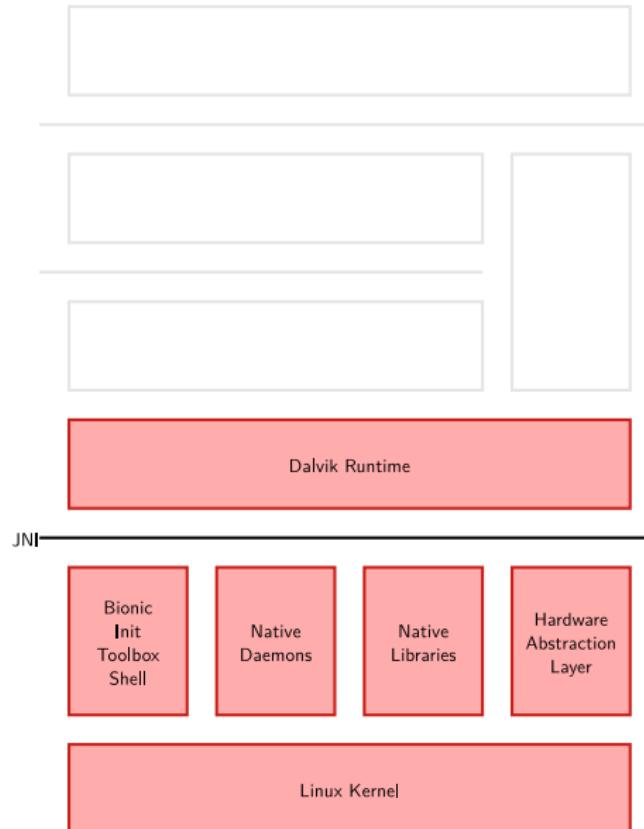


Android Native Layer

JNI



Whole Android Stack





What is JNI?

- ▶ A Java framework to call and be called by native applications written in other languages
- ▶ Mostly used for:
 - ▶ Writing Java bindings to C/C++ libraries
 - ▶ Accessing platform-specific features
 - ▶ Writing high-performance sections
- ▶ It is used extensively across the Android user space to interface between the Java Framework and the native daemons
- ▶ Since Gingerbread, you can develop apps in a purely native way, possibly calling Java methods through JNI



C Code

```
#include "jni.h"

JNIEXPORT void JNICALL Java_com_example_Print_print(JNIEnv *env,
                                                    jobject obj,
                                                    jstring javaString)
{
    const char *nativeString = (*env)->GetStringUTFChars(env,
                                                          javaString,
                                                          0);
    printf("%s", nativeString);
    (*env)->ReleaseStringUTFChars(env, javaString, nativeString);
}
```



JNI arguments

- ▶ Function prototypes are following the template:

```
JNIEXPORT jstring JNICALL Java_ClassName_MethodName  
    (JNIEnv*, jobject)
```
- ▶ `JNIEnv` is a pointer to the JNI Environment that we will use to interact with the virtual machine and manipulate Java objects within the native methods
- ▶ `jobject` contains a pointer to the calling object. It is very similar to `this` in C++



Types

- ▶ There is no direct mapping between C Types and JNI types
- ▶ You must use the JNI primitives to convert one to his equivalent
- ▶ However, there are a few types that are directly mapped, and thus can be used directly without typecasting:

Native Type	JNI Type
unsigned char	jboolean
signed char	jbyte
unsigned short	jchar
short	jshort
long	jint
long long	jlong
float	jfloat
double	jdouble



Java Code

```
package com.example;

class Print
{
    private static native void print(String str);

    public static void main(String[] args)
    {
        Print.print("HelloWorld!");
    }

    static
    {
        System.loadLibrary("print");
    }
}
```



Calling a method of a Java object from C

```
JNIEXPORT void JNICALL Java_ClassName_Method(JNIEnv *env,
                                              jobject obj)
{
    jclass cls = (*env)->GetObjectClass(env, obj);
    jmethodID hello = (*env)->GetMethodID(env,
                                             cls,
                                             "hello",
                                             "(V)V");
    if (!hello)
        return;
    (*env)->CallVoidMethod(env, obj, hello);
}
```



Instantiating a Java object from C

```
JNIEXPORT jobject JNICALL Java_ClassName_Method(JNIEnv *env,
                                                 jobject obj)
{
    jclass cls = env->FindClass("java/util/ArrayList");
    jmethodID init = env->GetMethodID(cls,
                                         "<init>",
                                         "()V");
    jobject array = env->NewObject(cls, init);

    return array;
}
```



Practical lab - Develop a JNI library

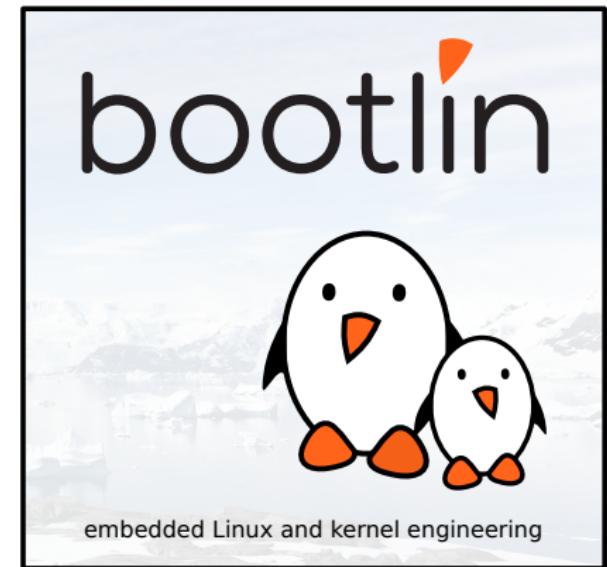


- ▶ Develop bindings from Java to C
- ▶ Integrate these bindings into the build system



Android Framework and Applications

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

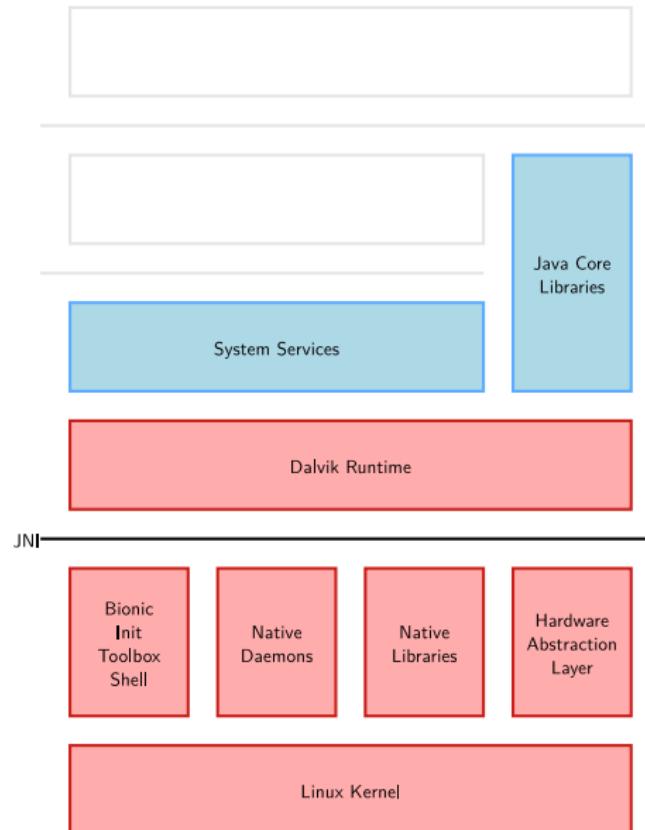




Service Manager and Various Services

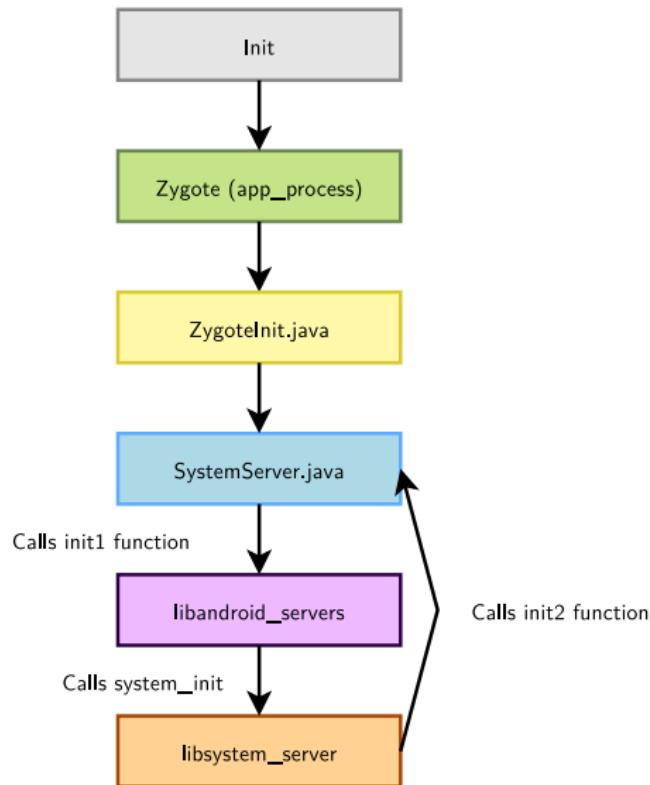


Whole Android Stack





System Server boot





The first step: system_server.c

- ▶ Located in frameworks/base/cmds/system_server
- ▶ Started by Zygote through the SystemServer
- ▶ Starts all the various native services:
 - ▶ SurfaceFlinger
 - ▶ SensorService
- ▶ It then calls back the SystemServer object's `init2` function to go on with the initialization



Java Services Initialization

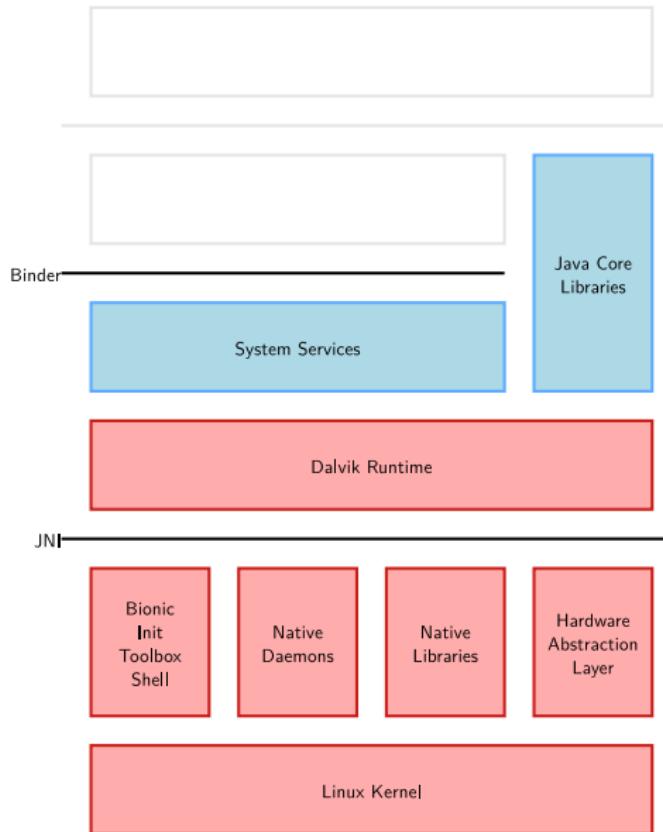
- ▶ Located in frameworks/base/services/java/com/android/server/SystemServer.java
- ▶ Starts all the different Java services in a different thread by registering them into the Service Manager
- ▶ PowerManager, ActivityManager (also handles the ContentProviders), PackageManager, BatteryService, LightsService, VibratorService, AlarmManager, WindowManager, BluetoothService, DevicePolicyManager, StatusBarManager, InputMethodManager, ConnectivityService, MountService, NotificationManager, LocationManager, AudioService,
...
▶ If you wish to add a new system service, you will need to add it to one of these two parts to register it at boot time



Inter-Process Communication, Binder and AIDLs



Whole Android Stack





- ▶ On modern systems, each process has its own address space, allowing to isolate data
- ▶ This allows for better stability and security: only a given process can access its address space. If another process tries to access it, the kernel will detect it and kill this process.
- ▶ However, interactions between processes are sometimes needed, that's what IPCs are for.
- ▶ On classic Linux systems, several IPC mechanisms are used:
 - ▶ Signals
 - ▶ Semaphores
 - ▶ Sockets
 - ▶ Message queues
 - ▶ Pipes
 - ▶ Shared memory
- ▶ Android, however, uses mostly:
 - ▶ Binder
 - ▶ Ashmem and Sockets



Binder 1/2

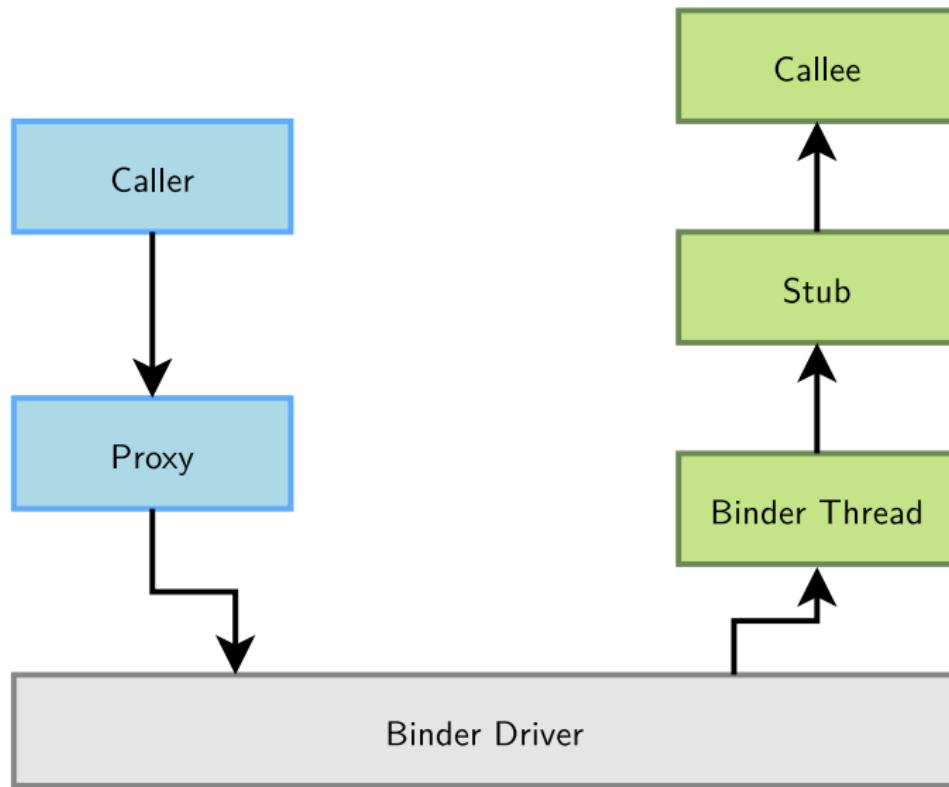
- ▶ Uses shared memory for high performance
- ▶ Uses reference counting to garbage collect objects no longer in use
- ▶ Data are sent through *parcels*, which is some kind of serialization
- ▶ Used across the whole system, e.g., clients connect to the window manager through Binder, which in turn connects to SurfaceFlinger using Binder
- ▶ Each object has an *identity*, which does not change, even if you pass it to other processes.



- ▶ This is useful if you want to separate components in distinct processes, or to manage several components of a single process (i.e. Activity's Windows).
- ▶ Object identity is also used for security. Some token passed correspond to specific permissions. Another security model to enforce permissions is for every transaction to check on the calling UID.
- ▶ Binder also supports one-way and two-way messages

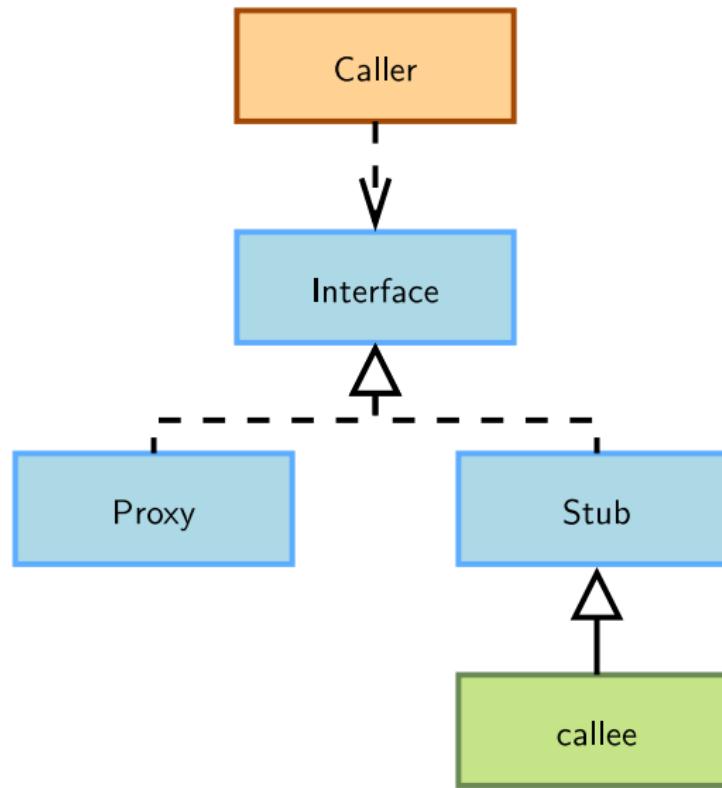


Binder Mechanism



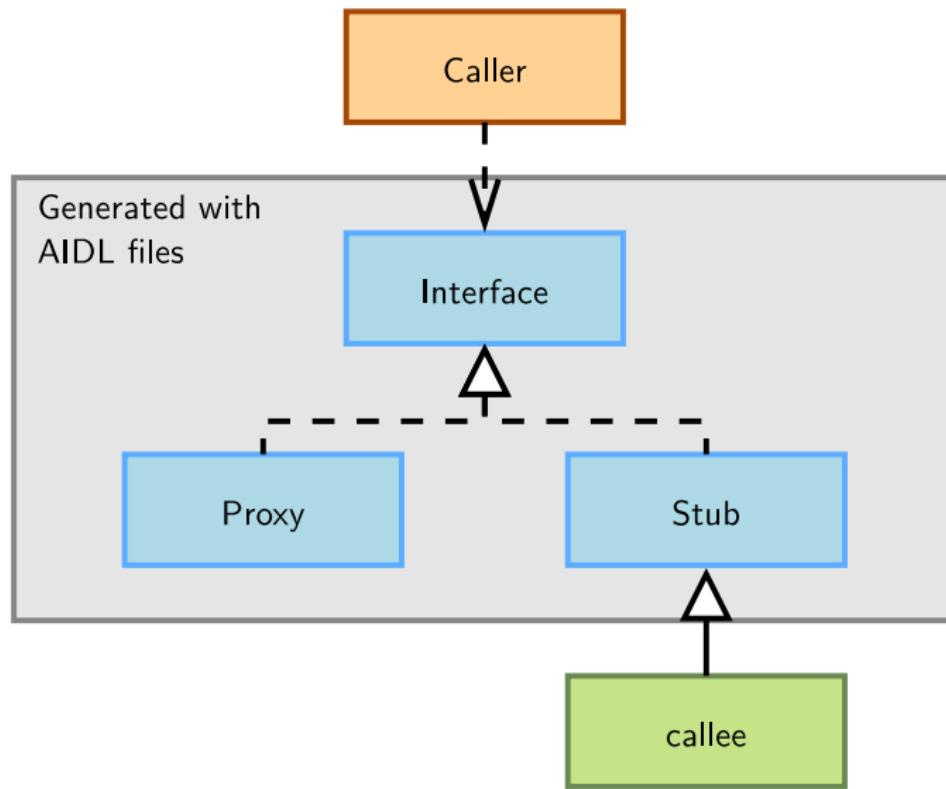


Binder Implementation 1/2





Binder Implementation 2/2





Android Interface Definition Language (AIDL)

- ▶ Very similar to any other Interface Definition Language you might have encountered
- ▶ Describes a programming interface for the client and the server to communicate using IPCs
- ▶ Looks a lot like Java interfaces. Several types are already defined, however, and you can't extend this like what you can do in Java:
 - ▶ All Java primitive types (int, long, boolean, etc.)
 - ▶ String
 - ▶ CharSequence
 - ▶ Parcelable
 - ▶ List of one of the previous types
 - ▶ Map



AIDLs HelloWorld

```
package com.example.android;  
  
interface IRemoteService {  
    void HelloPrint(String aString);  
}
```



Parcelable Objects

- ▶ If you want to add extra objects to the AIDLs, you need to make them implement the `Parcelable` interface
- ▶ Most of the relevant Android objects already implement this interface.
- ▶ This is required to let Binder know how to serialize and deserialize these objects
- ▶ However, this is not a general purpose serialization mechanism. Underlying data structures may evolve, so you should not store parcelled objects to persistent storage
- ▶ Has primitives to store basic types, arrays, etc.
- ▶ You can even serialize file descriptors!



Implement Parcelable Classes

- ▶ To make an object parcelable, you need to:
 - ▶ Make the object implement the `Parcelable` interface
 - ▶ Implement the `writeToParcel` function, which stores the current state of the object to a `Parcel` object
 - ▶ Add a static field called `CREATOR`, which implements the `Parcelable.Creator` interface, and takes a `Parcel`, deserializes the values and returns the object
 - ▶ Create an AIDL file that declares your new parcelable class
- ▶ You should also consider `Bundles`, that are type-safe key-value containers, and are optimized for reading and writing values



Intents

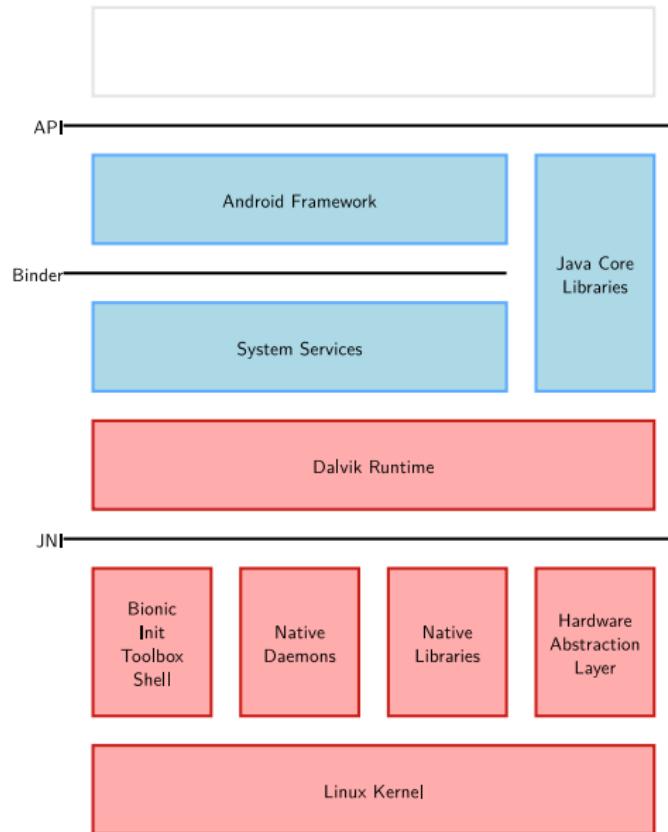
- ▶ Intents are a high-level use of Binder
- ▶ They describe the intention to do something
- ▶ They are used extensively across Android
 - ▶ Activities, Services and BroadcastReceivers are started using intents
- ▶ Two types of intents:
 - explicit** The developer designates the target by its name
 - implicit** There is no explicit target for the Intent. The system will find the best target for the Intent by itself, possibly asking the user what to do if there are several matches



Various Java Services



Whole Android Stack





- ▶ There are lots of services implemented in Java in Android
- ▶ They abstract most of the native features to make them available in a consistent way
- ▶ You get access to the system services using the `Context.getSystemService()` call
- ▶ You can find all the accessible services in the documentation for this function



ActivityManager

- ▶ Manages everything related to Android applications
 - ▶ Starts Activities and Services through Zygote
 - ▶ Manages their lifecycle
 - ▶ Fetches content exposed through content providers
 - ▶ Dispatches the implicit intents
 - ▶ Adjusts the Low Memory Killer priorities
 - ▶ Handles non responding applications



PackageManager

- ▶ Exposes methods to query and manipulate already installed packages, so you can:
 - ▶ Get the list of packages
 - ▶ Get/Set permissions for a given package
 - ▶ Get various details about a given application (name, uids, etc)
 - ▶ Get various resources from the package
- ▶ You can even install/uninstall an apk
 - ▶ `installPackage/uninstallPackage` functions are hidden in the source code, yet public.
 - ▶ You can't compile code that is calling directly these functions and they are not documented anywhere except in the code
 - ▶ But you can call them through the Java Reflection API, if you have the proper permissions of course



- ▶ Abstracts the Wakelocks functionality
- ▶ Defines several states, but when a wakelock is grabbed, the CPU will always be on
 - ▶ PARTIAL_WAKE_LOCK
 - ▶ Only the CPU is on, screen and keyboard backlight are off
 - ▶ SCREEN_DIM_WAKE_LOCK
 - ▶ Screen backlight is partly on, keyboard backlight is off
 - ▶ SCREEN_BRIGHT_WAKE_LOCK
 - ▶ Screen backlight is on, keyboard backlight is off
 - ▶ FULL_WAKE_LOCK
 - ▶ Screen and keyboard backlights are on



AlarmManager

- ▶ Abstracts the Android timers
- ▶ Allows to set a one time timer or a repetitive one
- ▶ When a timer expires, the AlarmManager grabs a wakelock, sends an Intent to the corresponding application and releases the wakelock once the Intent has been handled



ConnectivityManager and WifiManager

- ▶ ConnectivityManager

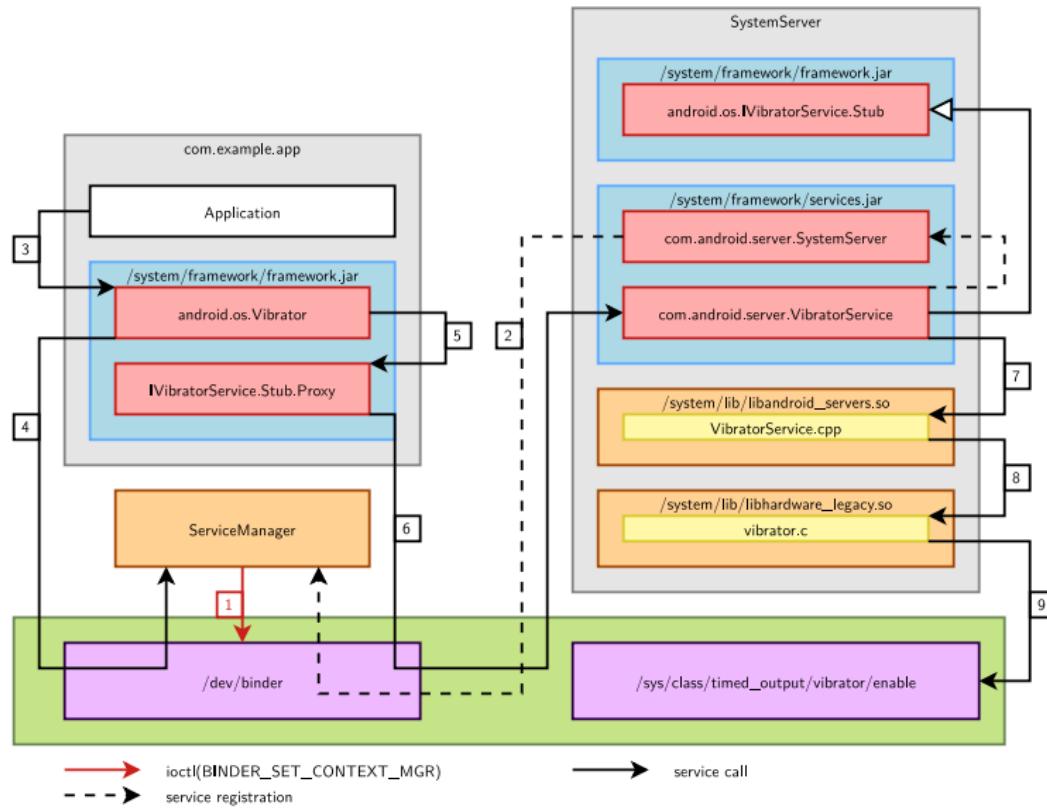
- ▶ Manages the various network connections
 - ▶ Falls back to other connections when one fails
 - ▶ Notifies the system when one becomes available/unavailable
 - ▶ Allows the applications to retrieve various information about connectivity

- ▶ WifiManager

- ▶ Provides an API to manage all aspects of WiFi networks
 - ▶ List, modify or delete already configured networks
 - ▶ Get information about the current WiFi network if any
 - ▶ List currently available WiFi networks
 - ▶ Sends Intents for every change in WiFi state



Example: Vibrator Service





Extend the framework



Why extend it?

- ▶ You might want to extend the existing Android framework to add new features or allow other applications to use specific devices available on your hardware
- ▶ As you have the code, you could just hack the source to make the framework suit your needs
- ▶ This is quite problematic however:
 - ▶ You might break the API, introduce bugs, etc
 - ▶ Google requires you not to modify the Android public API
 - ▶ It is painful to track changes across the tree, to port the changes to new versions
 - ▶ You don't always want to have such extensions for all your products
- ▶ As usual with Android, there's a device-specific way of extending the framework: PlatformLibraries



- ▶ The modifications are just plain Java libraries
- ▶ You can declare any namespace you want, do whatever code you want.
- ▶ However, they are bundled as raw Java archives, so you cannot embed resources in the modifications
- ▶ If you would still do this, you can add them to frameworks/base/res, but you have to hide them
- ▶ When using the Google Play Store, all the libraries including these ones are submitted to Google, so that it can filter out apps relying on libraries not available on your system
- ▶ To avoid any application to link to any jar file, you have to declare both in your application and in your library that you will use and add a custom library
- ▶ The library's xml permission file should go into the /system/etc/permissions folder



PlatformLibrary Makefile

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES := \
    $(call all-subdir-jar-files)

LOCAL_MODULE_TAGS := optional

LOCAL_MODULE:= com.example.android.pl

include $(BUILD_JAVA_LIBRARY)
```



PlatformLibrary permissions file

```
<?xml version="1.0" encoding="utf-8"?>
<permissions>
    <library name="com.example.android.pl"
        file="/system/framework/com.example.android.pl.jar"/>
</permissions>
```



PlatformLibrary Client Makefile

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional

LOCAL_PACKAGE_NAME := PlatformLibraryClient

LOCAL_SRC_FILES := $(call all-java-files-under, src)

LOCAL_JAVA_LIBRARIES := com.example.android.pl

include $(BUILD_PACKAGE)
```



Practical lab - Develop a Framework Component

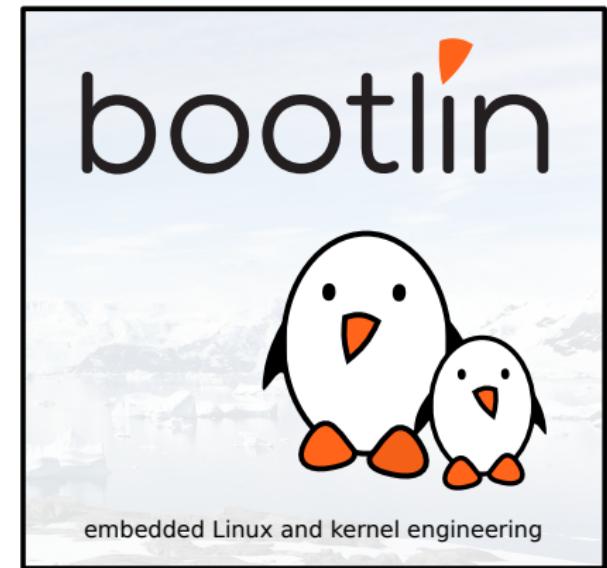


- ▶ Modify the Android framework
- ▶ Use JNI bindings



Android Application Development

© Copyright 2004-2019, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

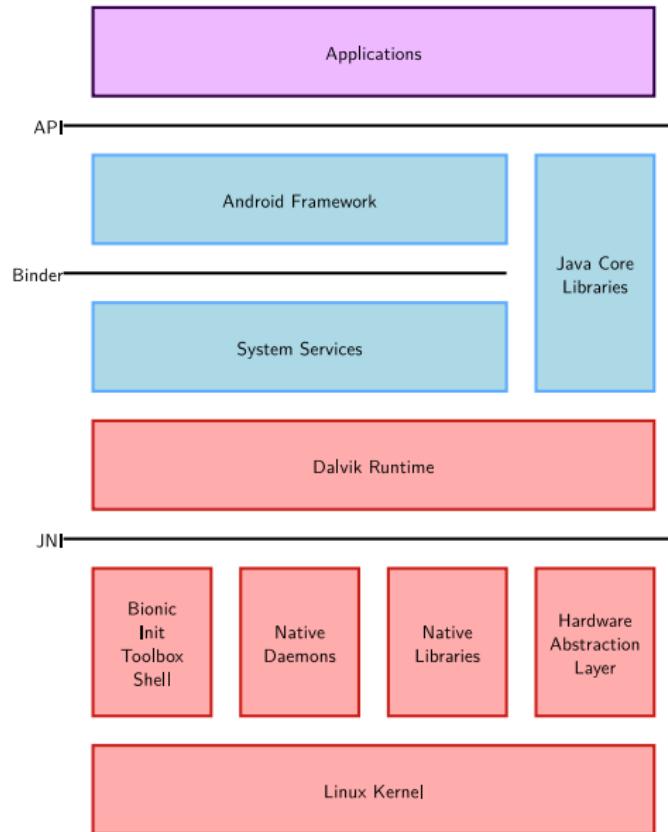




Basics



Whole Android Stack





Android applications

- ▶ Android applications are written mostly in Java using Google's SDK
- ▶ Applications are bundled into an Android PacKage (.apk files) which are archives containing the compiled code, data and resources for the application, so applications are completely self-contained
- ▶ You can install applications either through a market (Google Play Store, Amazon Appstore, F-Droid, etc) or manually (through ADB or a file manager)
- ▶ Of course, everything we have seen so far is mostly here to provide a nice and unified environment to application developers



Applications Security

- ▶ Once installed, applications live in their own sandbox, isolated from the rest of the system
- ▶ The system assigns a Linux user to every application, so that every application has its own user/group
- ▶ It uses this UID and files permissions to allow the application to access only its own files
- ▶ Each process has its own instance of Dalvik, so code is running isolated from other applications
- ▶ By default, each application runs in its own process, which will be started/killed during system life
- ▶ Android uses the *principle of least privilege*. Each application by default has only access to what it requires to work.
- ▶ However, you can request extra permissions, make several applications run in the same process, or with the same UID, etc.



Applications Components

- ▶ Components are the basic blocks of each application
- ▶ You can see them as entry points for the system in the application
- ▶ There are four types of components:
 - ▶ Activities
 - ▶ Broadcast Receivers
 - ▶ Content Providers
 - ▶ Services
- ▶ Every application can start any component, even located in other applications. This allows to share components easily, and have very little duplication. However, for security reasons, you start it through an Intent and not directly
- ▶ When an application requests a component, the system starts the process for this application, instantiates the needed class and runs that component. We can see that there is no single point of entry in an application like `main()`



Application Manifest

- ▶ To declare the components present in your application, you have to write a XML file, `AndroidManifest.xml`
- ▶ This file is used to:
 - ▶ Declare available components
 - ▶ Declare which permissions these components need
 - ▶ Revision of the API needed
 - ▶ Declare hardware features needed
 - ▶ Libraries required by the components



Manifest HelloWorld

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.example.android">
    <application>
        <activity android:name=".ExampleActivity"
                  android:label="@string/example_label">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <uses-library android:name="com.example.android.pl" />
    </application>
</manifest>
```



- ▶ Google also provides a NDK to allow developers to write native code
- ▶ While the code is not run by Dalvik, the security guarantees are still there
- ▶ Allows to write faster code or to port existing C code to Android more easily
- ▶ Since Gingerbread, you can even code a whole application without writing a single line of Java
- ▶ It is still packaged in an apk, with a manifest, etc.
- ▶ However, there are some drawbacks, the main one being that you can't access the resources mechanism available from Java



Activities

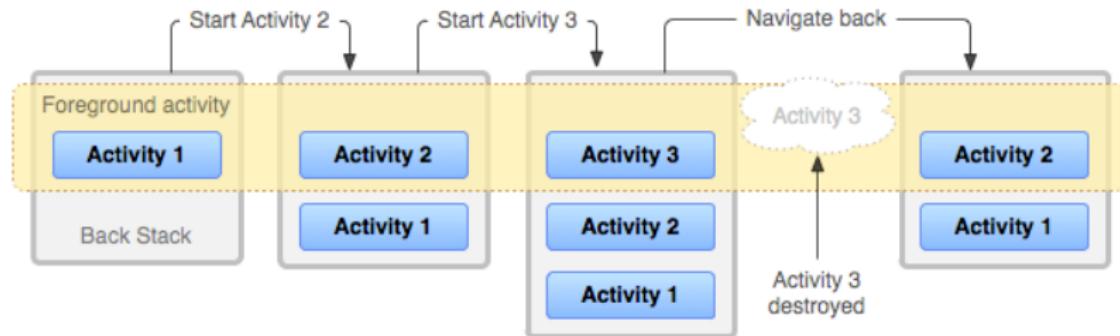


Activities

- ▶ Activities are a single screen of the user interface of an application
- ▶ They are assembled to provide a consistent interface. If we take the example of an email application, we will have:
 - ▶ An activity listing the received mails
 - ▶ An activity to compose a new mail
 - ▶ An activity to read a mail
- ▶ Other applications might need one of these activities. To continue with this example, the Camera application might want to start the composing activity to share the just-shot picture
- ▶ It is up to the application developer to advertise available activities to the system
- ▶ When an activity starts a new activity, the latter replaces the former on the screen and is pushed on the *back stack* which holds the last used activities, so when the user is done with the newer activity, it can easily go back to the previous one



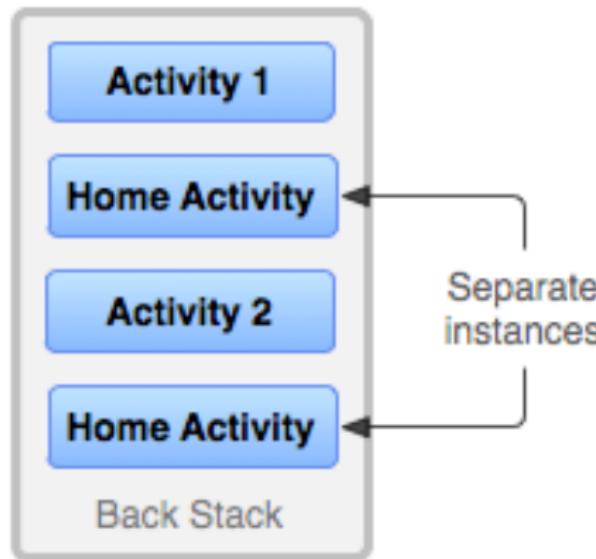
Back Stack



Credits: <http://developer.android.com>



Back Stack



Credits: <http://developer.android.com>



Activity Lifecycle 1/3

- ▶ As there is no single entry point and as the system manages the activities, activities have to define callbacks that the system can call at some point in time
- ▶ Activities can be in one of the three states on Android
 - Running** The activity is on the foreground and has focus
 - Paused** The activity is still visible on the screen but no longer has focus. It can be destroyed by the system under very heavy memory pressure
 - Stopped** The activity is no longer visible on the screen. It can be killed at any time by the system

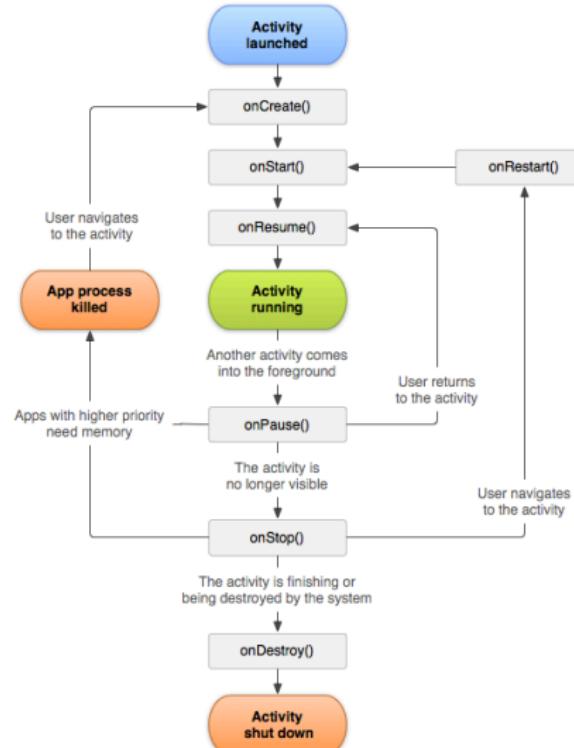


Activity Lifecycle 2/3

- ▶ There are callbacks for every change from one of these states to another
- ▶ The most important ones are `onCreate` and `onPause`
- ▶ All components of an application run in the same thread. If you do long operations in the callbacks, you will block the entire application (UI included). You should always use threads for every long-running task.



Activity Lifecycle 3/3



Credits: <http://developer.android.com>



Saving Activity State 1/2

- ▶ As applications tend to be killed and restarted quite often, we need a way to store our internal state when killed and reload it when restarted
- ▶ Once again, this is done through callbacks
- ▶ Before killing the application, the system calls the `onSaveInstanceState` callback and when restarting it, it calls `onRestoreInstanceState`
- ▶ In both cases, it provides a Bundle as argument to allow the activity to store what's needed and reload it later, with little overhead

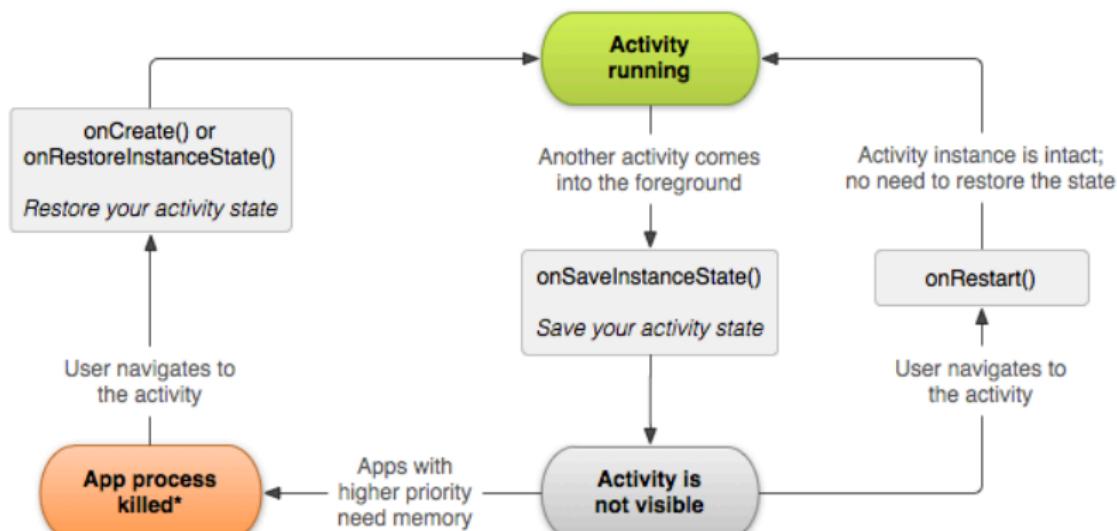


Saving Activity State 2/2

- ▶ This makes the creation/suppression of activities flawless for the user, while allowing to save as much memory as we need
- ▶ These callbacks are not always called though. If the activity is killed because the user left it in a permanent way (through the back button), it won't be called
- ▶ By default, these activities are also called when rotating the device, because the activity will be killed and restarted by the system to load new resources



Activity Lifecycle

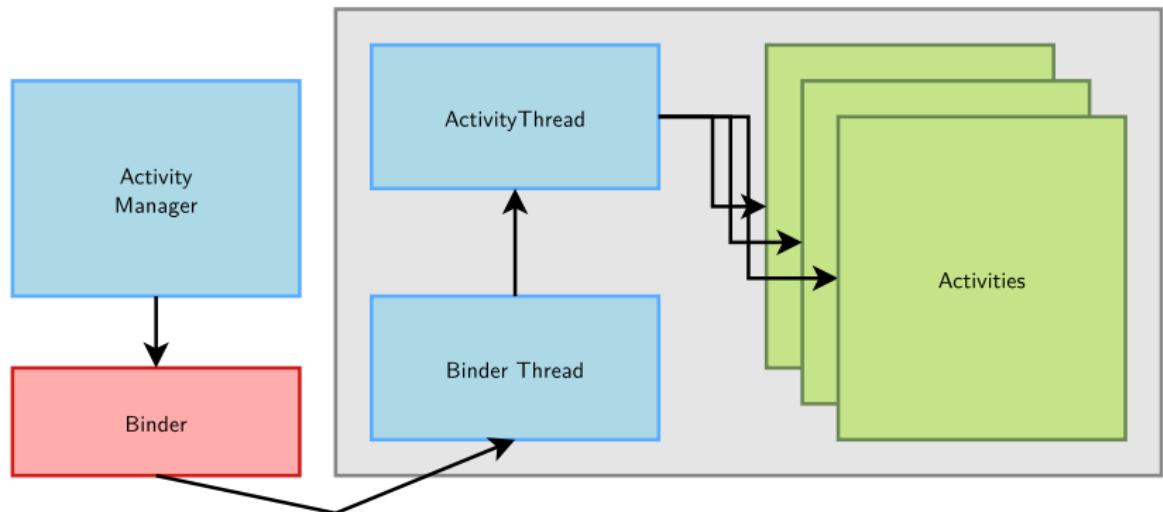


*Activity instance is destroyed, but the state from `onSaveInstanceState()` is saved

Credits: <http://developer.android.com>



Activity Callbacks



Credits: <http://developer.android.com>



Activity HelloWorld

```
public class ExampleActivity extends Activity {  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.example);  
        Log.i("ExampleActivity", "Activity created!");  
    }  
    protected void onStart() {  
        super.onStart();  
    }  
    protected void onResume() {  
        super.onResume();  
    }  
    protected void onPause() {  
        super.onPause();  
    }  
    protected void onStop() {  
        super.onStop();  
    }  
    protected void onDestroy() {  
        super.onDestroy();  
    }  
}
```



Services



Services

- ▶ Services are components running in the background
- ▶ They are used either to perform long running operations or to work for remote processes
- ▶ A service has no user interface, as it is supposed to run when the user does something else
- ▶ From another component, you can either work with a service in a synchronous way, by *binding* to it, or asynchronous, by *starting* it



Service Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.example.android">
    <application>
        <service android:name=".ExampleService"/>
    </application>
</manifest>
```

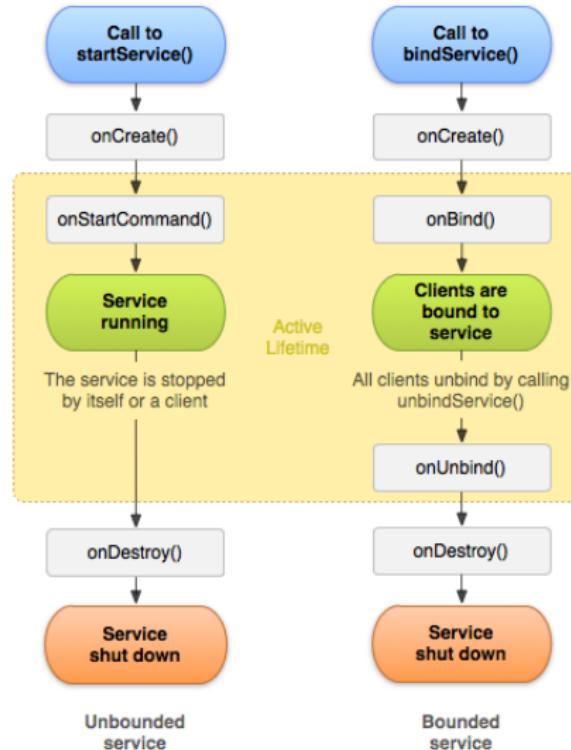


Services Types

- ▶ We can see services as a set including:
 - ▶ Started Services, that are created when other components call `startService`. Such a service runs as long as needed, whether the calling component is still alive or not, and can stop itself or be stopped. When the service is stopped, it is destroyed by the system
 - ▶ You can also subclass `IntentService` to have a started service. However, while much easier to implement, this service will not handle multiple requests simultaneously.
 - ▶ Bound Services, that are bound to by other components by calling `bindService`. They offer a client/server like interface, interacting with each other. Multiple components can bind to it, and a service is destroyed only when no more components are bound to it
- ▶ Services can be of both types, given that callbacks for these two do not overlap completely
- ▶ Services are started by passing Intents either to the `startService` or `bindService` commands



Services Lifecycle



Credits: <http://developer.android.com>

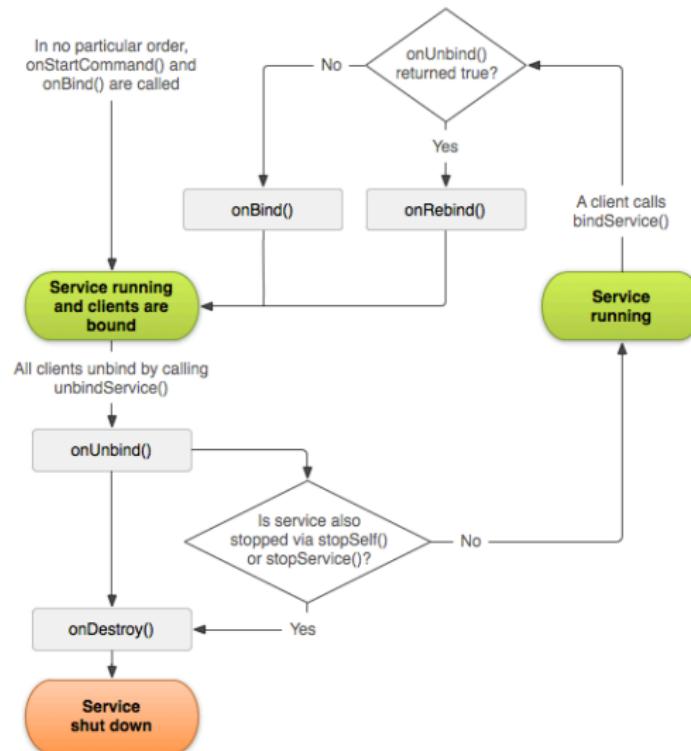


Bound Services

- ▶ There are three possible ways to implement a bound service:
 - ▶ By extending the `Binder` class. It works only when the clients are local and run in the same process though.
 - ▶ By using a `Messenger`, that will provide the interface for your service to remote processes. However, it does not perform multi-threading, all requests are queued up.
 - ▶ By writing your own `aidl` file. You will then be able to implement your own interface and write thread-safe code, as you are very likely to receive multiple requests at once



Bound Services and Started Lifecycle



Credits: <http://developer.android.com>



Content Providers



Content Providers

- ▶ They provide access to organized data in a manner quite similar to relational databases
- ▶ They allow to share data with both internal and external components and centralize them
- ▶ Security is also enforced by permissions like usual, but they also do not allow remote components to issue arbitrary requests like what we can do with relational databases
- ▶ Instead, Content Providers rely on URIs to allow for a restricted set of requests with optional parameters, only permitting the user to filter by values and by columns
- ▶ You can use any storage back-end you want, while exposing a quite neutral and consistent interface to other applications



Content URIs

- ▶ URIs are often built with the following pattern:
 - ▶ `content://<package>.provider/<path>` to access particular tables
 - ▶ `content://<package>.provider/<path>/<id>` to access single rows inside the given table
- ▶ Facilities are provided to deal with these
 - ▶ On the application side:
 - ▶ `ContentUri` to append and manage numerical IDs in URIs
 - ▶ `Uri.Builder` and `Uri` classes to deal with URIs and strings
 - ▶ On the provider side:
 - ▶ `UriMatcher` associates a pattern to an ID, so that you can easily match incoming URIs, and use switch over them.



Implementing a Content Provider

```
public class ExampleProvider extends ContentProvider {
    private static final UriMatcher sUriMatcher;

    static {
        sUriMatcher.addURI("com.example.android.provider", "table1", 1);
        sUriMatcher.addURI("com.example.android.provider", "table1#", 2);
    }

    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder) {

        switch (sUriMatcher.match(uri)) {
            default:
                System.out.println("Hello World!");
                break;
        }
    }
}
```



Implementing a Content Provider

```
public Uri insert(Uri uri, ContentValues values) {
    return null;
}

public int update(Uri uri, ContentValues values, String selection,
                  String[] selectionArgs) {
    return 0;
}

public int delete(Uri uri, String selection, String[] selectionArgs) {
    return 0;
}

public boolean onCreate() {
    return true;
}
```



Managing the Intents



Intents

- ▶ Intents are basically a bundle of several pieces of information, mostly
 - ▶ Component Name
 - ▶ Contains both the full class name of the target component plus the package name defined in the Manifest
 - ▶ Action
 - ▶ The action to perform or that has been performed
 - ▶ Data
 - ▶ The data to act upon, written as a URI, like
`tel://0123456789`
 - ▶ Category
 - ▶ Contains additional information about the nature of the component that will handle the intent, for example the launcher or a preference panel
- ▶ The component name is optional. If it is set, the intent will be explicit. Otherwise, the intent will be implicit



Intent Resolution

- ▶ When using explicit intents, dispatching is quite easy, as the target component is explicitly named. However, it is quite rare that a developer knows the component name of external applications, so it is mostly used for internal communication.
- ▶ Implicit intents are a bit more tricky to dispatch. The system must find the best candidate for a given intent.
- ▶ To do so, components that want to receive intents have to declare them in their manifests *Intent filters*, so that the system knows what components it can respond to.
- ▶ Components without intent filters will never receive implicit intents, only explicit ones



Intent Filters 1/2

- ▶ They are only about notifying the system about handled implicit intents
- ▶ Filters are based on matching by category, action and data. Filtering by only one of these three (by category for example) is fine.
 - ▶ A filter can list several actions. If an intent action field corresponds to one of the actions listed here, the intent will match
 - ▶ It can also list several categories. However, if none of the categories of an incoming intent are listed in the filter, then intent won't match.



Intent Filters 2/2

- ▶ You can also use intent matching from your application by using the `query*` methods from the PackageManager to get a matching component from an Intent.
- ▶ For example, the launcher application does that to display only activities with filters that specify the category `android.intent.category.LAUNCHER` and the action `android.intent.action.MAIN`



Real Life Manifest Example: Notepad

```
<manifest package="com.example.android.notepad">
    <application android:icon="@drawable/app_notes"
                  android:label="@string/app_name" >

        <activity android:name="NotesList"
                  android:label="@string/title_notes_list">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <action android:name="android.intent.action.EDIT" />
                <action android:name="android.intent.action.PICK" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```



Broadcasted intents

- ▶ Intents can also be broadcast thanks to two functions:
 - ▶ `sendBroadcast` that broadcasts an intent that will be handled by all its handlers at the same time, in an undefined order
 - ▶ `sendOrderedBroadcast` broadcasts an intent that will be handled by one handler at a time, possibly with propagation of the result to the next handler, or the possibility for a handler to cancel the broadcast
- ▶ Broadcasts are used for system wide notification of important events: booting has completed, a package has been removed, etc.



Broadcast Receivers

- ▶ Broadcast receivers are the fourth type of components that can be integrated into an application. They are specifically designed to deal with broadcast intents.
- ▶ Their overall design is quite easy to understand: there is only one callback to implement: `onReceive`
- ▶ The lifecycle is quite simple too: once the `onReceive` callback has returned, the receiver is considered no longer active and can be destroyed at any moment
- ▶ Thus you must not use asynchronous calls (Bind to a service for example) from the `onReceive` callback, as there is no way to be sure that the object calling the callback will still be alive in the future.



Processes and Threads



Process Management in Android

- ▶ By default in Android, every component of a single application runs in the same process.
- ▶ When the system wants to run a new component:
 - ▶ If the application has no running component yet, the system will start a new process with a single thread of execution in it
 - ▶ Otherwise, the component is started within that process
- ▶ If you happen to want a component of your application to run in its own process, you can still do it through the `android:process` XML attribute in the manifest.
- ▶ When the memory constraints are high, the system might decide to kill a process to get some memory back. This is done based on the importance of the process to the user. When a process is killed, all the components running inside are killed.



Processes priority

- ▶ *Foreground processes* have the topmost priority. They host either
 - ▶ An activity the user is interacting with
 - ▶ A service bound to such an activity
 - ▶ A service running in the foreground (started with `startForeground`)
 - ▶ A service running one of its lifecycle callbacks
 - ▶ A broadcast receiver running its `onReceive` method
- ▶ *Visible processes* host
 - ▶ An activity that is no longer in the foreground but still is visible on the screen
 - ▶ A service that is bound to a visible activity
- ▶ *Service Processes* host a service that has been started by `startService`
- ▶ *Background Processes* host activities that are no longer visible to the user
- ▶ *Empty Processes*



Threads

- ▶ As there is only one thread of execution, both the application components and UI interactions are done in sequential order
- ▶ So a long computation, I/O, background tasks cannot be run directly into the main thread without blocking the UI
- ▶ If your application is blocked for more than 5 seconds, the system will display an “*Application Not Responding*” dialog, which leads to poor user experience
- ▶ Moreover, UI functions are not thread-safe in Android, so you can only manipulate the UI from the main thread.
- ▶ So, you should:
 - ▶ Dispatch every long operation either to a service or a worker thread
 - ▶ Use messages between the main thread and the worker threads to interact with the UI.



Threads in Android

- ▶ There are two ways of implementing worker threads in Android:
 - ▶ Use the standard Java threads, with a class extending `Runnable`
 - ▶ This works, of course, but you will need to do messaging between your worker thread and the main thread, either through handlers or through the `View.post` function
 - ▶ Use Android's `AsyncTask`
 - ▶ A class that has four callbacks: `doInBackground`, `onPostExecute`, `onPreExecute`, `onProgressUpdate`
 - ▶ Useful, because only `doInBackground` is called from a worker thread, others are called by the UI thread



Resources



Applications Resources

- ▶ Applications contain more than just compiled source code: images, videos, sound, etc.
- ▶ In Android, anything related to the visual appearance of the application is kept separate from the source code: activities layout, animations, menus, strings, etc.
- ▶ Resources should be kept in the `res/` directory of your application.
- ▶ At compilation, the build tool will create a class `R`, containing references to all the available resources, and associating an ID to it
- ▶ This mechanism allows you to provide several alternatives to resources, depending on locales, screen size, pixel density, etc. in the same application, resolved at runtime.

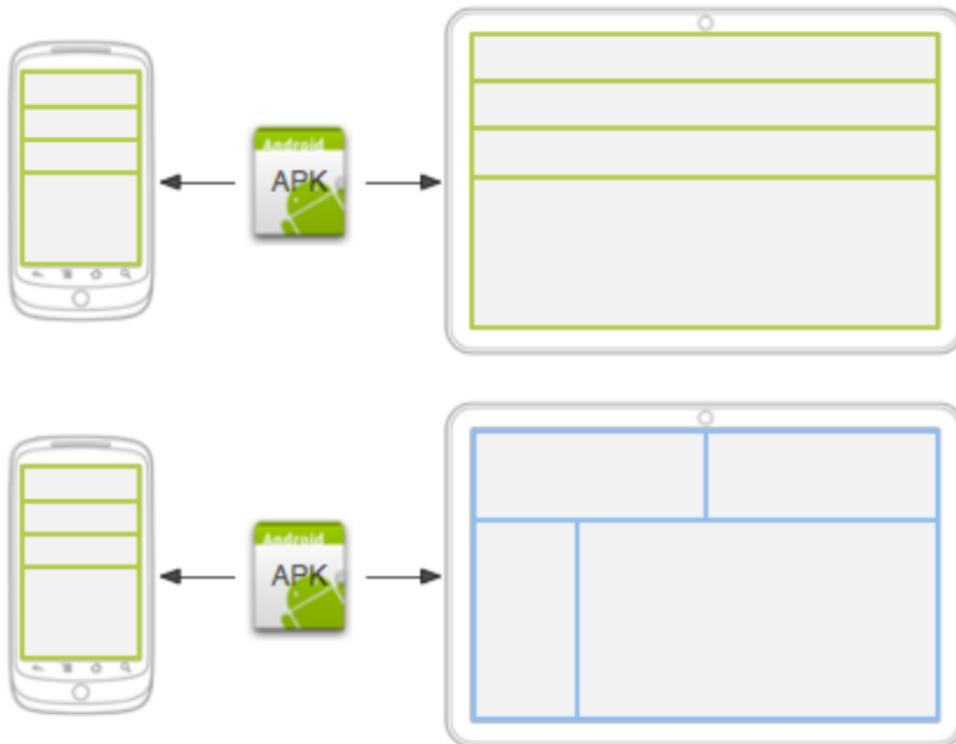


Resources Directory

- ▶ All resources are located in the `res/` subdirectory
 - ▶ `anim/` contains animation definitions
 - ▶ `color/` contains the color definitions
 - ▶ `drawable/` contains images, "9-patch" graphics, or XML-files defining drawables (shapes, widgets, relying on a image file)
 - ▶ `layout/` contains XML defining applications layout
 - ▶ `menu/` contains XML files for the menu layouts
 - ▶ `raw/` contains files that are left untouched
 - ▶ `values/` contains strings, integers, arrays, dimensions, etc
 - ▶ `xml/` contains arbitrary XML files
- ▶ All these files are accessed by applications through their IDs.
If you still want to use a file path, you need to use the
`assets/` folders



Resources



Credits: <http://developer.android.com>

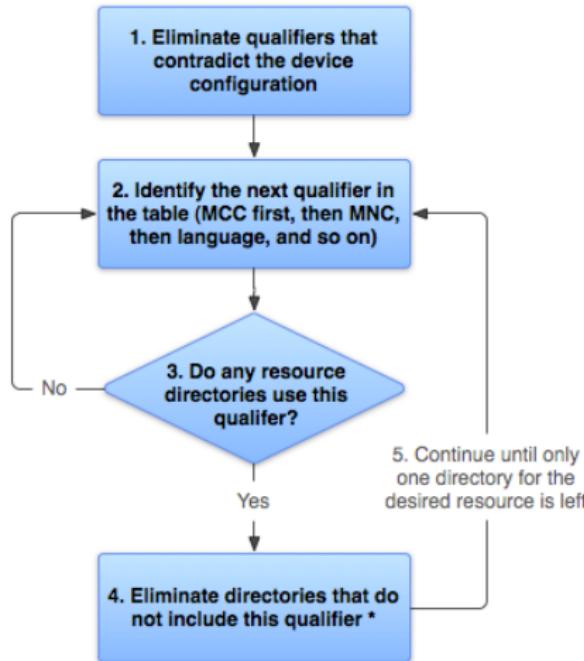


Alternative Resources

- ▶ Alternative resources are provided using extended sub-folder names, that should be named using the pattern
`<folder_name>-<qualifier>`
- ▶ There is a number of qualifiers, depending on which case you want to provide an alternative for. The most used ones are probably:
 - ▶ locales (en, fr, fr-rCA, ...)
 - ▶ screen orientation (land, port)
 - ▶ screen size (small, large,...)
 - ▶ screen density (mdpi, ldpi, ...)
 - ▶ and much others
- ▶ You can specify multiple qualifiers by chaining them, separated by dashes. If you want layouts to be applied only when on landscape on high density screens, you will save them into the directory `layout-land-hdpi`



Resources Selection



* If the qualifier is screen density, the system selects the "best match" and the process is done

Credits: <http://developer.android.com>



Data Storage



Data Storage on Android

- ▶ An application might need to write to arbitrary files and read from them, for caching purposes, to make settings persistent, etc.
- ▶ But the system can't just let you read and write to any random file on the system, this would be a major security flaw
- ▶ Android provides some mechanisms to address the two following concerns: allow an application to write to files, while integrating it into the Android security model
- ▶ There are four major mechanisms:
 - ▶ Preferences
 - ▶ Internal data
 - ▶ External data
 - ▶ Databases



Shared Preferences

- ▶ Shared Preferences allows to store and retrieve data in a persistent way
- ▶ They are stored using key-value pairs, but can only store basic types: int, float, string, boolean
- ▶ They are persistent, so you don't have to worry about them disappearing when the activity is killed
- ▶ You can get an instance of the class managing the preferences through the function `getPreferences`
- ▶ You may also want several set of preferences for your application and the function `getSharedPreferences` for that
- ▶ You can edit them by calling the method `edit` on this instance. Don't forget to call `commit` when you're done!



Internal Storage

- ▶ You can also save files directly to the internal storage device
- ▶ These files are not accessible by default by other applications
- ▶ Such files are deleted when the user removes the application
- ▶ You can request a `FileOutputStream` class to such a new file by calling the method `openFileOutput`
- ▶ You can pass extra flags to this method to either change the way the file is opened or its permissions
- ▶ These files will be created at runtime. If you want to have files at compile time, use resources instead
- ▶ You can also use internal storage for caching purposes. To do so, call `getCacheDir` that will return a `File` object allowing you to manage the cache folder the way you want to. Cache files may be deleted by Android when the system is low on internal storage.



External Storage

- ▶ External storage is either the SD card or an internal storage device
- ▶ Each file stored on it is world-readable, and the user has direct access to it, since that is the device exported when USB mass storage is used.
- ▶ Since this storage may be removable, your application should check for its presence, and that it behaves correctly
- ▶ You can either request a sub-folder created only for your application using the `getExternalFilesDir` method, with a tag giving which type of files you want to store in this directory. This folder will be removed at un-installation.
- ▶ Or you can request a public storage space, shared by all applications, and never removed by the system, using `getExternalStoragePublicDirectory`
- ▶ You can also use it for caching, with `getExternalCacheDir`



SQLite Databases

- ▶ Databases are often abstracted by Content Providers, that will abstract requests, but Android adds another layer of abstraction
- ▶ Databases are managed through subclasses of `SQLiteOpenHelper` that will abstract the structure of the database
- ▶ It will hold the requests needed to build the tables, views, triggers, etc. from scratch, as well as requests to migrate to a newer version of the same database if its structure has to evolve.
- ▶ You can then get an instance of `SQLiteDatabase` that allows to query the database
- ▶ Databases created that way will be only readable from your application, and will never be automatically removed by the system
- ▶ You can also manipulate the database using the `sqlite3` command in the shell



Android Packages (apk)

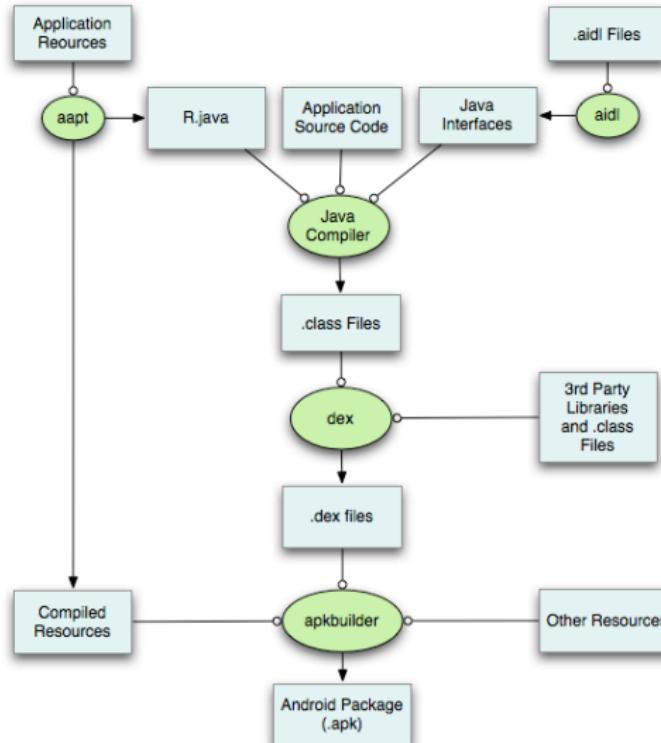


Content of an APK

- ▶ META-INF a directory containing all the Java metadata
 - ▶ MANIFEST.MF the Java Manifest file, containing various metadata about the classes present in the archive
 - ▶ CERT.RSA Certificate of the application
 - ▶ CERT.SF List of resources present in the package and associated SHA-1 hash
- ▶ AndroidManifest.xml
- ▶ res contains all the resources, compiled to binary xml for the relevant resources
- ▶ classes.dex contains the compiled Java classes, to the *Dalvik EXecutable* format, which is a uncompressed format, containing Dalvik instructions
- ▶ resources.arsc is the resources table. It keeps track of the package resources, associated IDs and packages



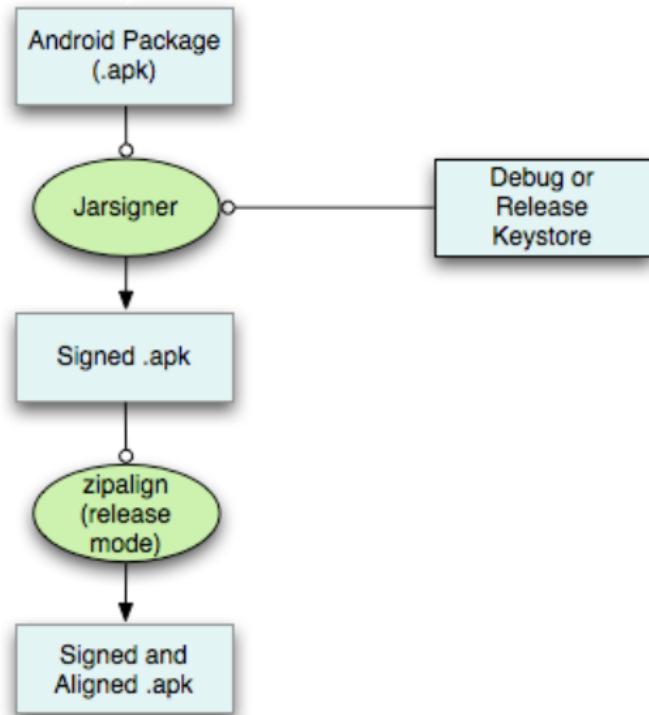
APK Building



Credits: <http://developer.android.com>



APK Building



Credits: <http://developer.android.com>



Practical lab - Write an Application with the SDK



- ▶ Write an Android application
- ▶ Integrate an application in the
Android build system



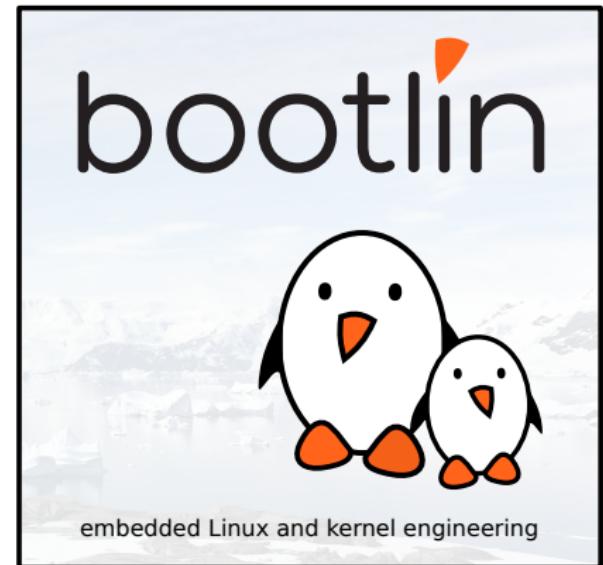
Advices and Resources

Advices and Resources

© Copyright 2004-2019, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!

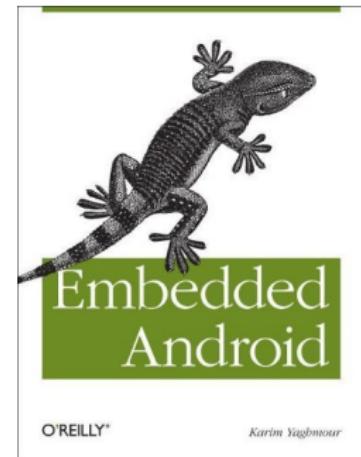




Android Internals

Embedded Android: Porting, Extending, and Customizing, April 2013

- ▶ By Karim Yaghmour, O'Reilly
- ▶ From what we know from the preview version, good reference book and guide on all hidden and undocumented Android internals
- ▶ Our rating: 3 stars

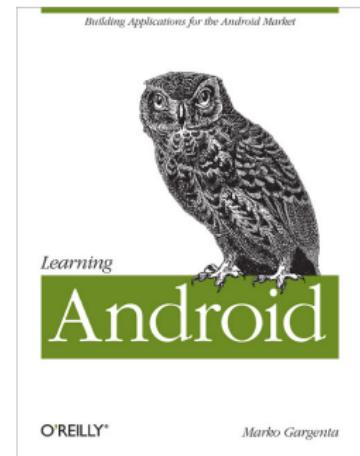




Android Development

Learning Android, March 2011

- ▶ By Marko Gargenta, O'Reilly
- ▶ A good reference book and guide on
Android application development
- ▶ Our rating: 2 stars





Websites

- ▶ Android API reference:
<http://developer.android.com/reference>
- ▶ Android Documentation:
<http://developer.android.com/guide/>
- ▶ A good overview on how the various parts of the system are put together to maintain a highly secure system
<http://source.android.com/tech/security/>



Conferences

Useful conferences featuring Android topics:

- ▶ Embedded Linux Conference:

<http://embeddedlinuxconference.com/>

Organized by the Linux Foundation in the USA (Spring) and in Europe (Fall). Mostly about kernel and user space Linux development in general, but always some talks about Android. Presentation slides and videos freely available



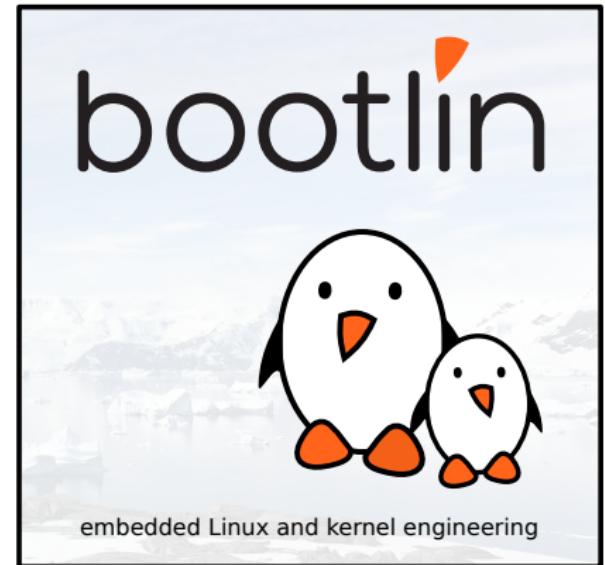
Last slides

Last slides

© Copyright 2004-2019, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Thank you!
And may the Source be with you