Android System Development

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux
Developers
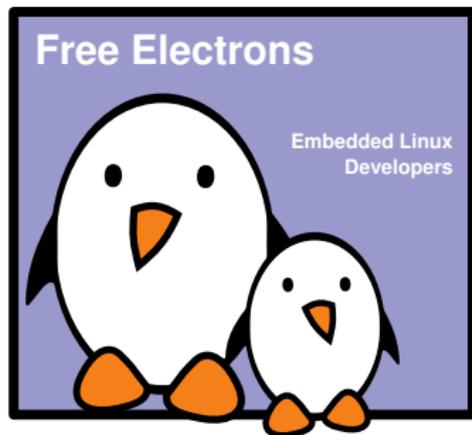
Introduction to Android

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux
Developers
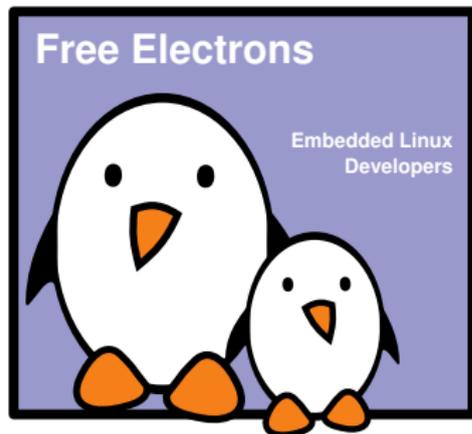
Features

- ▶ 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 WebKit rendering engine
    - ▶ Support for hardware accelerated graphics through OpenGL ES
    - ▶ Support for all the common wireless mechanisms: GSM, CDMA, UMTS, LTE, Bluetooth, WiFi.

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

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, the Panda board and the Motorola Xoom.

Android Releases

- Each new version is given a dessert name
- Released in alphabetical order
- Last releases:
    - Android 2.3 Gingerbread
    - Android 3.X Honeycomb
    - Android 4.0 Ice Cream Sandwich

1.0
"Apple Pie"

↓

1.1
"Banana Bread"

↓

1.5
"Cupcake"

↓

1.6
"Donut"

↓

2.0/2.1
"Eclair"

↓

2.2
"Frozen Yogurt"

↓

2.3
"Gingerbread"

3.0/3.1/3.2
"Honeycomb"

4.0
"Ice Cream Sandwich"

Phones only

Tablets only

Tablets and phones
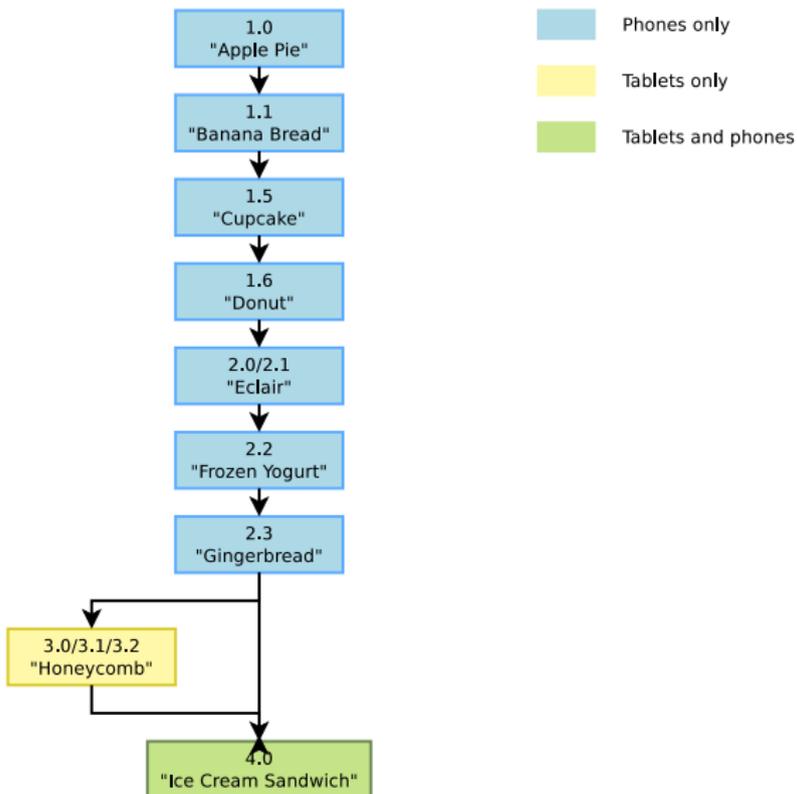
Architecture

The Linux Kernel

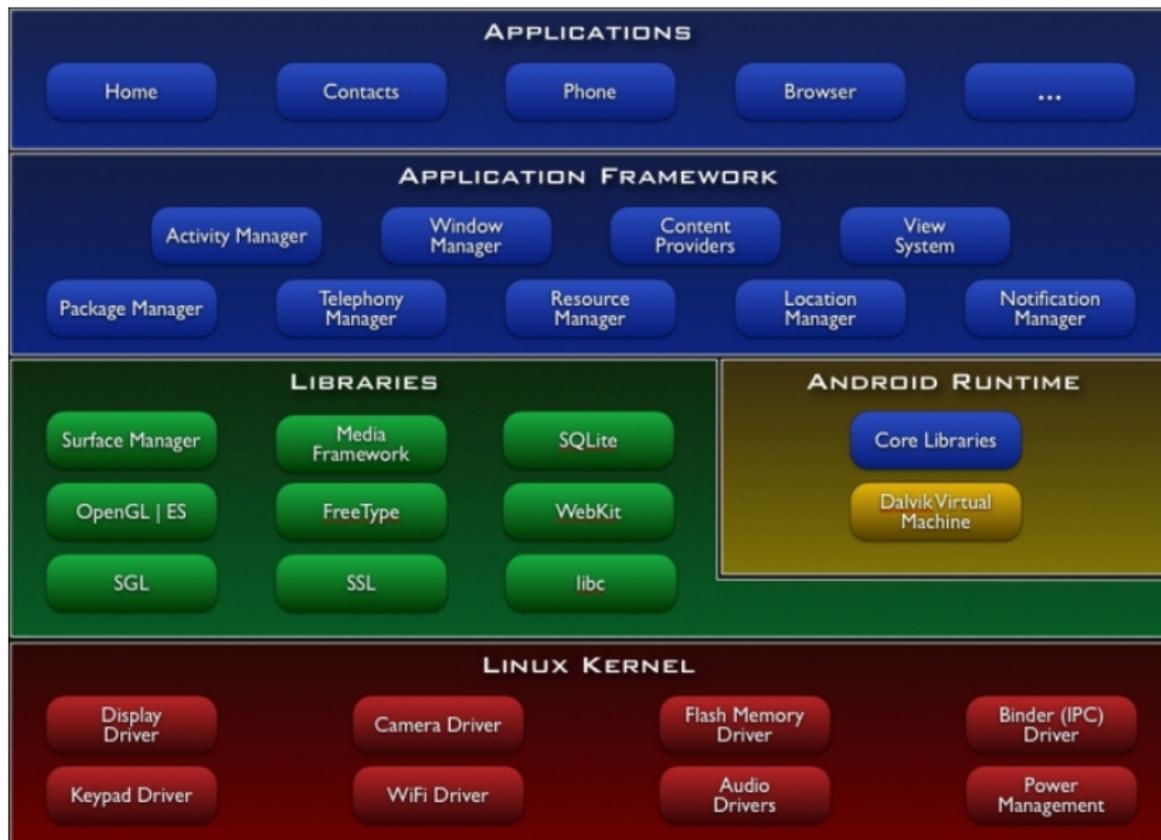- Used as the foundation of the Android system
- Numerous additions from the stock Linux, including new IPC (Inter-Process Communication) mechanisms, alternative power management mechanism, new drivers and various additions across the kernel
- These changes are beginning to go into the `staging/` area 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 WebKit, 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.

- 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.

Android Source Code and Compilation

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux Developers
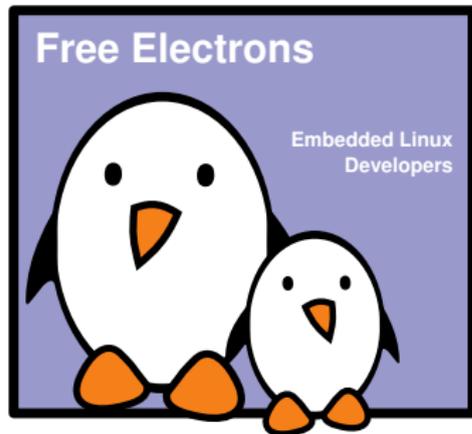
How to get the source code

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

Repo

- ▶ 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

Source code licenses

- Mostly two kind of licenses:
  - GPL/LGPL Code: Linux, D-Bus, BlueZ
  - Apache/BSD: All the rest
  - In the `external` folder, it depends on the component, but mostly GPL
- 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.

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

external/ is one of the largest folder of 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

libcore/ is the Java core library

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.

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 to 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 current selected product
- There are lots of 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
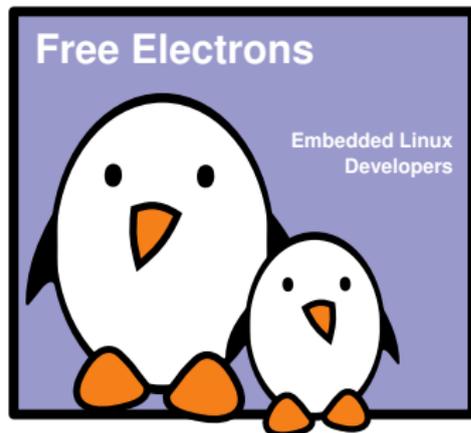
Contribute

Gerrit

- ▶ Google also developed for the Android development process 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`

Linux kernel introduction

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux
Developers

Linux features

User app B

Library A

User app A

C library

Call to services

Event notification,
information exposition

Linux kernel

Manage hardware

Event notification

Hardware
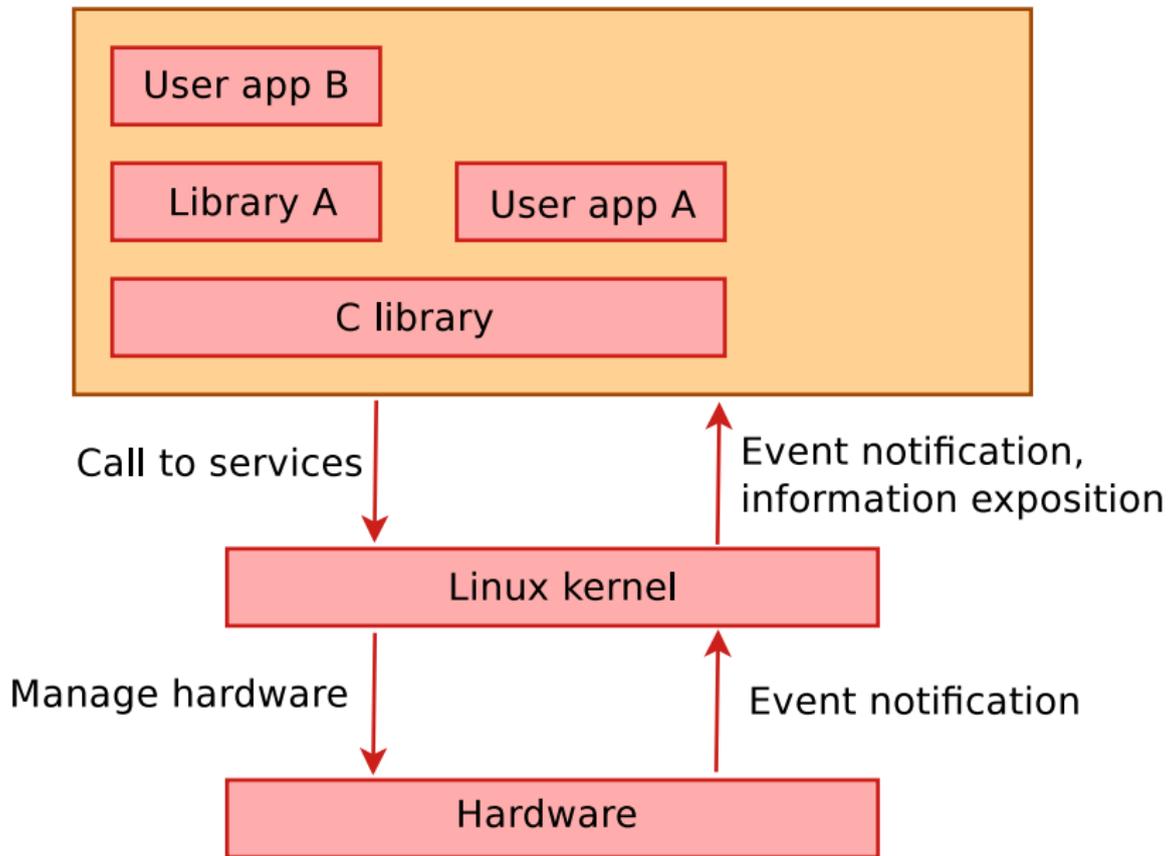
History

- 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, hundreds of people contribute to each kernel release, individuals or companies big and small.

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..

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.

Supported hardware architectures

3.0 status

- ▶ See the `arch/` directory in the kernel sources
- ▶ Minimum: 32 bit processors, with or without MMU, and `gcc` support
- ▶ 32 bit architectures (`arch/` subdirectories)
  `arm, avr32, blackfin, cris, frv, h8300, m32r, m68k, microblaze, mips, mn10300, parisc, s390, score, sparc, um, unicore32, xtensa`
- ▶ 64 bit architectures:
  `alpha, ia64, sparc64, tile`
- ▶ 32/64 bit architectures
  `powerpc, x86, sh`
- ▶ Find details in kernel sources: `arch/<arch>/Kconfig`, `arch/<arch>/README`, or `Documentation/<arch>/`

System calls

- ▶ The main interface between the kernel and userspace is the set of system calls
- ▶ About 300 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 userspace applications usually never make a system call directly but rather use the corresponding C library function

Virtual filesystems

- ▶ Linux makes system and kernel information available in user-space through virtual filesystems.
- ▶ Virtual filesystems allow applications to see directories and files that do not exist on any real storage: they are created on the fly by the kernel
- ▶ The two most important virtual filesystems are
  - ▶ `proc`, for process-related information
  - ▶ `sysfs`, for device-related information

# 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.
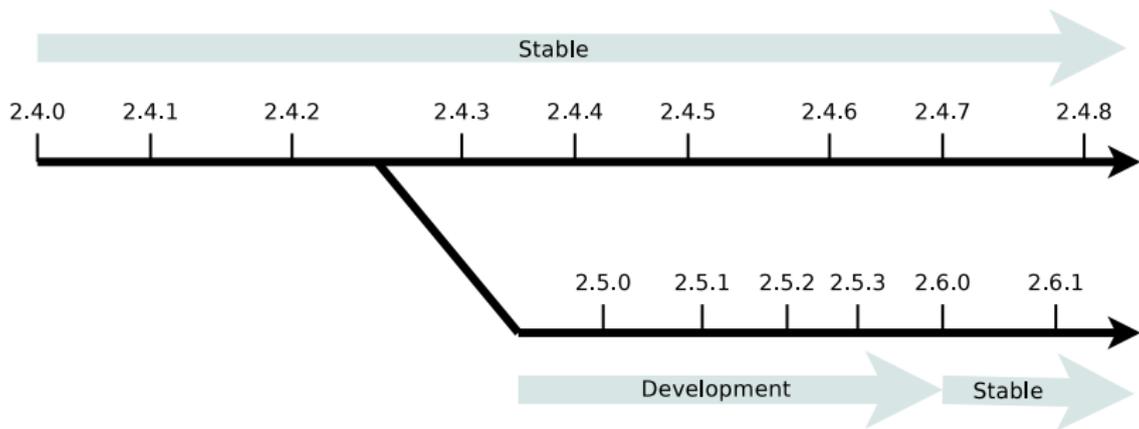
- ▶ Since `2.6.0`, kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make major changes in existing subsystems.
- ▶ So far, there was no need to create a new development branch (such as `2.7`), which would massively break compatibility with the stable branch.
- ▶ Thanks to this, **more features are released to users at a faster pace**.

Changes since Linux 2.6 (2)

Since 2.6.14, the kernel developers agreed on the following development model:

- ▶ After the release of a `2.6.x` version, a two-weeks merge window opens, during which major additions are merged.
- ▶ The merge window is closed by the release of test version `2.6.(x+1)-rc1`
- ▶ The bug fixing period opens, for 6 to 10 weeks.
- ▶ At regular intervals during the bug fixing period, `2.6.(x+1)-rcY` test versions are released.
- ▶ When considered sufficiently stable, kernel `2.6.(x+1)` is released, and the process starts again.
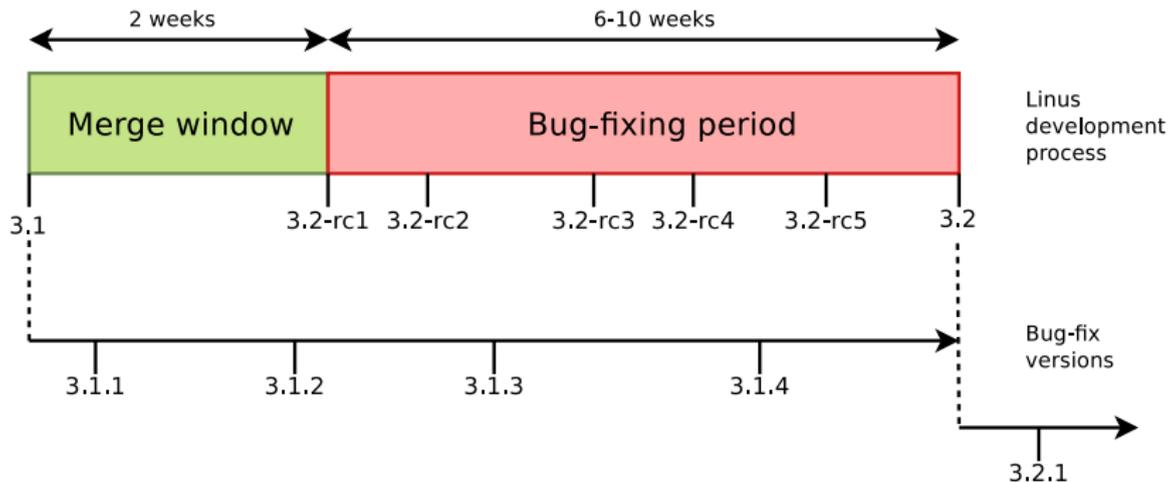
More stability for the 2.6 kernel tree

- ▶ Issue: bug and security fixes only released for most recent stable kernel versions.
- ▶ Some people need to have a recent kernel, but with long term support for security updates.
- ▶ You could get long term support from a commercial embedded Linux provider.
- ▶ You could reuse sources for the kernel used in Ubuntu Long Term Support releases (5 years of free security updates).
- ▶ You could choose one of the versions advertised as "long term" in the `http://kernel.org` front page. They will be maintained longer (2 or 3 years), unlike other versions.

linux-next:	**next-20110118**
snapshot:	**2.6.37-git18**
mainline:	**2.6.37**
stable:	**2.6.37**
stable:	**2.6.36.3**
longterm:	**2.6.35.10**
stable:	**2.6.35.9**
longterm:	**2.6.34.8**
stable:	**2.6.34.7**
stable:	**2.6.33.7**
longterm:	**2.6.32.28**
stable:	**2.6.32.28**
longterm:	**2.6.27.57**
stable:	**2.6.27.57**
stable:	**2.4.37.11**

New 3.x branch

- From 2003 to 2011, the official kernel versions were named `2.6.x`.
- Linux `3.0` was released in July 2011
- There is no change to the development model, only a change to the numbering scheme
    - Official kernel versions will be named `3.x` (`3.0`, `3.1`, `3.2`, etc.)
    - Stabilized versions will be named `3.x.y` (`3.0.2`, `3.4.3`, etc.)
    - It effectively only removes a digit compared to the previous numbering scheme

What's new in each Linux release?

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

```
commit aa6e52a35d388e730f4df0ec2ec48294590cc459
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.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 those 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@free-electrons.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.

- ▶ Fortunately, there are some useful resources available
  - ▶ http://wiki.kernelnewbies.org/LinuxChanges
  - ▶ http://lwn.net
  - ▶ http://linuxfr.org, for French readers

Kernel configuration

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-2.6.x/`
    - `make <target>`

Kernel configuration (1)

- ▶ 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 your hardware (for device drivers, etc.)
  - ▶ On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.)

Kernel configuration (2)

- ▶ The configuration is stored in the `.config` file at the root of kernel sources
  - ▶ Simple text file, `key=value` style
- ▶ 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
- ▶ To modify a kernel in a GNU/Linux distribution: the configuration files are usually released in `/boot/`, together with kernel images: `/boot/config-2.6.17-11-generic`

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**
  - ▶ Those 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
    - `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
  - ▶ `make xconfig` allows to see all options, even those that cannot be selected because of missing dependencies. In this case, they are displayed in gray

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: `libqt4-dev`

make xconfig screenshot

make xconfig search interface

Looks for a keyword in the parameter name. Allows to select or unselect found parameters.
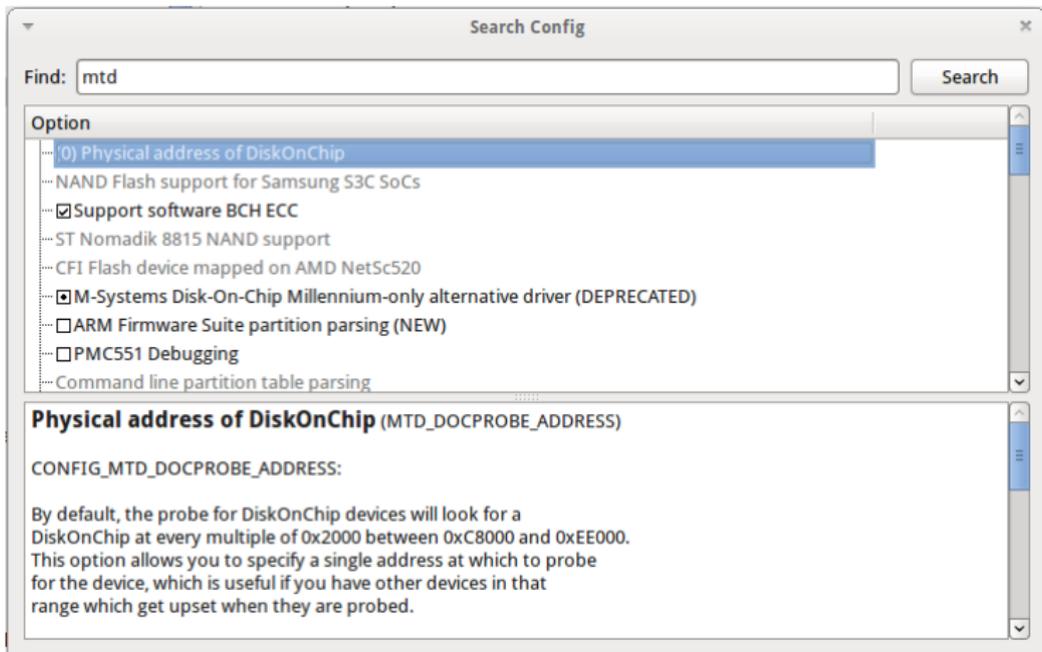
Compiled as a module (separate file)
`CONFIG_ISO9660_FS=m`

Driver options
`CONFIG_JOLIET=y`

`CONFIG_ZISOFS=y`

⊡ ISO 9660 CDROM file system support
  ☑ Microsoft Joliet CDROM extensions
  ☑ Transparent decompression extension
⊡ UDF file system support

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`
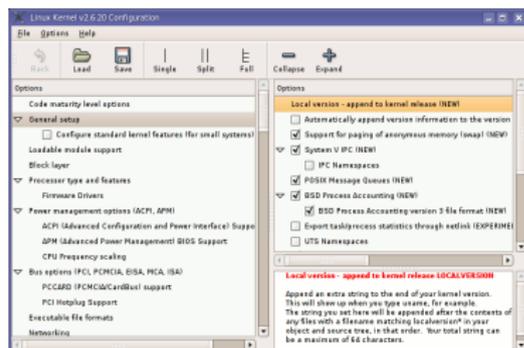
make menuconfig

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`

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

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

make allnoconfig

make allnoconfig

- ▶ Only sets strongly recommended settings to y.
- ▶ Sets all other settings to n.
- ▶ Very useful in embedded systems to select only the minimum required set of features and drivers.
- ▶ Much more convenient than unselecting hundreds of features one by one!

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, allnoconfig...) keep this .config.old backup copy.

Configuration per architecture

- The set of configuration options is architecture dependent
  - Some configuration options are very architecture-specific
  - Most of the configuration options (global kernel options, network subsystem, filesystems, most of the device drivers) are visible in all architectures.
- By default, the kernel build system assumes that the kernel is being built for the host architecture, i.e. native compilation
- The architecture is not defined inside the configuration, but at an higher level
- We will see later how to override this behaviour, to allow the configuration of kernels for a different architecture

Overview of kernel options (1)

- General setup
  - *Prompt for development/incomplete code* allows to be able to enable drivers or features that are not considered as completely stable yet
  - *Local version - append to kernel release* allows to concatenate an arbitrary string to the kernel version that an user can get using `uname -r`. Very useful for support!
  - *Support for swap*, can usually be disabled on most embedded devices
  - *Configure standard kernel features (for small systems)* allows to remove features from the kernel to reduce its size. Powerful, use with care!

Overview of kernel options (2)

- ▶ Loadable module support
  - ▶ Allows to enable or completely disable module support. If your system doesn't need kernel modules, best to disable since it saves a significant amount of space and memory
- ▶ Enable the block layer
  - ▶ If CONFIG_EXPERT is enabled, the block layer can be completely removed. Embedded systems using only Flash storage can safely disable the block layer
- ▶ Processor type and features (x86) or System type (ARM) or CPU selection (MIPS)
  - ▶ Allows to select the CPU or machine for which the kernel must be compiled
  - ▶ On x86, only optimization-related, on other architectures very important since there's no compatibility

- ▶ Kernel features
    - ▶ Tickless system, which allows to disable the regular timer tick and use on-demand ticks instead. Improves power savings
    - ▶ High resolution timer support. By default, the resolution of timer is the tick resolution. With high resolution timers, the resolution is as precise as the hardware can give
    - ▶ Preemptible kernel enables the preemption inside the kernel code (the userspace code is always preemptible). See our real-time presentation for details
- ▶ Power management
    - ▶ Global power management option needed for all power management related features
    - ▶ Suspend to RAM, CPU frequency scaling, CPU idle control, suspend to disk

Overview of kernel options (4)

- ▶ Networking support
    - ▶ The network stack
    - ▶ Networking options
        - ▶ Unix sockets, needed for a form of inter-process communication
        - ▶ TCP/IP protocol with options for multicast, routing, tunneling, Ipsec, Ipv6, congestion algorithms, etc.
        - ▶ Other protocols such as DCCP, SCTP, TIPC, ATM
        - ▶ Ethernet bridging, QoS, etc.
    - ▶ Support for other types of network
        - ▶ CAN bus, Infrared, Bluetooth, Wireless stack, WiMax stack, etc.

Overview of kernel options (5)

- Device drivers
    - MTD is the subsystem for Flash (NOR, NAND, OneNand, battery-backed memory, etc.)
    - Parallel port support
    - Block devices, a few misc block drivers such as loopback, NBD, etc.
    - ATA/ATAPI, support for IDE disk, CD-ROM and tapes. A new stack exists
    - SCSI
        - The SCSI core, needed not only for SCSI devices but also for USB mass storage devices, SATA and PATA hard drives, etc.
        - SCSI controller drivers

Overview of kernel options (6)

- Device drivers (cont)
    - SATA and PATA, the new stack for hard disks, relies on SCSI
    - RAID and LVM, to aggregate hard drivers and do replication
    - Network device support, with the network controller drivers. Ethernet, Wireless but also PPP
    - Input device support, for all types of input devices: keyboards, mice, joysticks, touchscreens, tablets, etc.
    - Character devices, contains various device drivers, amongst them
        - serial port controller drivers
        - PTY driver, needed for things like SSH or telnet
    - I2C, SPI, 1-wire, support for the popular embedded buses
    - Hardware monitoring support, infrastructure and drivers for thermal sensors

Overview of kernel options (7)

- ▶ Device drivers (cont)
  - ▶ Watchdog support
  - ▶ Multifunction drivers are drivers that do not fit in any other category because the device offers multiple functionality at the same time
  - ▶ Multimedia support, contains the V4L and DVB subsystems, for video capture, webcams, AM/FM cards, DVB adapters
  - ▶ Graphics support, infrastructure and drivers for framebuffers
  - ▶ Sound card support, the OSS and ALSA sound infrastructures and the corresponding drivers
  - ▶ HID devices, support for the devices that conform to the HID specification (Human Input Devices)

- ► Device drivers (cont)
    - ► USB support
        - ► Infrastructure
        - ► Host controller drivers
        - ► Device drivers, for devices connected to the embedded system
        - ► Gadget controller drivers
        - ► Gadget drivers, to let the embedded system act as a mass-storage device, a serial port or an Ethernet adapter
    - ► MMC/SD/SDIO support
    - ► LED support
    - ► Real Time Clock drivers
    - ► Voltage and current regulators
    - ► Staging drivers, crappy drivers being cleaned up

- ▶ For some categories of devices the driver is not implemented inside the kernel
  - ▶ Printers
  - ▶ Scanners
  - ▶ Graphics drivers used by X.org
  - ▶ Some USB devices
- ▶ For these devices, the kernel only provides a mechanism to access the hardware, the driver is implemented in userspace

Overview of kernel options (10)

- ▶ File systems
  - ▶ The common Linux filesystems for block devices: ext2, ext3, ext4
  - ▶ Less common filesystems: XFS, JFS, ReiserFS, GFS2, OCFS2, Btrfs
  - ▶ CD-ROM filesystems: ISO9660, UDF
  - ▶ DOS/Windows filesystems: FAT and NTFS
  - ▶ Pseudo filesystems: proc and sysfs
  - ▶ Miscellaneous filesystems, with amongst other Flash filesystems such as JFFS2, UBIFS, SquashFS, cramfs
  - ▶ Network filesystems, with mainly NFS and SMB/CIFS
- ▶ Kernel hacking
  - ▶ Debugging features useful for kernel developers

Compiling and installing the kernel for the host system

Kernel compilation

- `make`
    - in the main kernel source directory
    - Remember to run `make -j 4` if you have multiple CPU cores to speed up the compilation process
    - No need to run as root !
- Generates
    - `vmlinux`, the raw uncompressed kernel image, at 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, `vmImage.gz` for Blackfin, etc.
    - All kernel modules, spread over the kernel source tree, as `.ko` files.

Kernel installation

- `make install`
    - Does the installation for the host system by default, so needs to be run as root. Generally not used when compiling for an embedded system, and it installs files on the development workstation.
- Installs
    - `/boot/vmlinuz-<version>`
      Compressed kernel image. Same as the one in `arch/<arch>/boot`
    - `/boot/System.map-<version>`
      Stores kernel symbol addresses
    - `/boot/config-<version>`
      Kernel configuration for this version
- Typically re-runs the bootloader configuration utility to take into account the new kernel.

Module installation

- `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`
      Module aliases for module loading utilities. Example line:
      `alias sound-service-?-0 snd_mixer_oss`
    - `modules.dep`
      Module dependencies
    - `modules.symbols`
      Tells which module a given symbol belongs to.

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: also removes your .config file!
  `make mrproper`

- Also remove editor backup and patch reject files (mainly to generate patches):
  `make distclean`

Cross-compiling the kernel

Cross-compiling the kernel

When you compile a Linux kernel for another CPU architecture

- ▶ Much faster than compiling natively, when the target system is much slower than your GNU/Linux workstation.
- ▶ Much easier as development tools for your GNU/Linux workstation are much easier to find.
- ▶ 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-`

Specifying cross-compilation

The CPU architecture and cross-compiler prefix are defined through the ARCH and CROSS_COMPILE variables in the toplevel Makefile.

▶ ARCH is the name of the architecture. It is defined by the name of the subdirectory in arch/ in the kernel sources
▶ CROSS_COMPILE is the prefix of the cross compilation tools
  ▶ Example: arm-linux- if your compiler is arm-linux-gcc
▶ Three solutions
  ▶ Force these two variables in the main kernel Makefile
    ARCH?= arm
    CROSS_COMPILE?= arm-linux-
  ▶ Pass ARCH and CROSS_COMPILE on the make command line
  ▶ Define ARCH and CROSS_COMPILE as environment variables
  ▶ Don't forget to have the values properly set at all steps, otherwise the kernel configuration and build system gets confused

Predefined configuration files

- Default configuration files available, per board or per-CPU family
    - They are stored in arch/<arch>/configs/, and are just minimal .config files
    - This is the most common way of configuring a kernel for embedded platforms
- Run make help to find if one is available for your platform
- To load a default configuration file, just run make acme_defconfig
    - This will overwrite your existing .config !
- To create your own default configuration file
    - make savedefconfig, to create a minimal configuration file
    - mv defconfig arch/<arch>/configs/myown_defconfig

Configuring the kernel

- ▶ After loading a default configuration file, you can adjust the configuration to your needs with the normal `xconfig`, `gconfig` or `menuconfig` interfaces
- ▶ You can also start the configuration from scratch without loading a default configuration file
- ▶ As the architecture is different than your host architecture
  - ▶ Some options will be different from the native configuration (processor and architecture specific options, specific drivers, etc.)
  - ▶ Many options will be identical (filesystems, network protocol, architecture-independent drivers, etc.)
- ▶ Make sure you have the support for the right CPU, the right board and the right device drivers.

Building and installing the kernel

- Run `make`
- Copy the final kernel image to the target storage
  - can be `uImage`, `zImage`, `vmlinux`, `bzImage` in `arch/<arch>/boot`
- make install is rarely used in embedded development, as the kernel image is a single file, easy to handle
  - It is however possible to customize the make install behaviour in `arch/<arch>/boot/install.sh`
- `make modules_install` is used even in embedded development, as it installs many modules and description files
  - `make INSTALL_MOD_PATH=<dir>/ modules_install`
  - The `INSTALL_MOD_PATH` variable is needed to install the modules in the target root filesystem instead of your host root filesystem.

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
  - ▶ and many more, documented in `Documentation/kernel-parameters.txt` in the kernel sources
- ▶ 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
  - ▶ Built into the kernel, using the `CONFIG_CMDLINE` option.

Changes introduced in the Android Kernel

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux
Developers
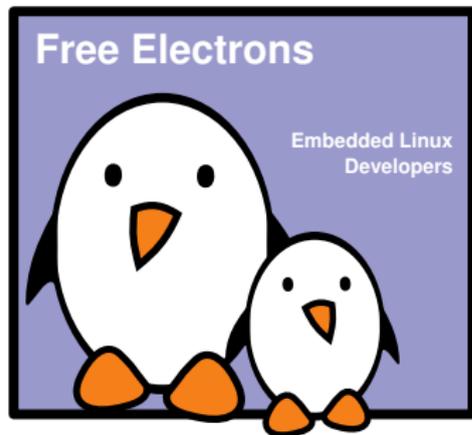
Wakelocks

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 deciding of when the devices needs 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

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.
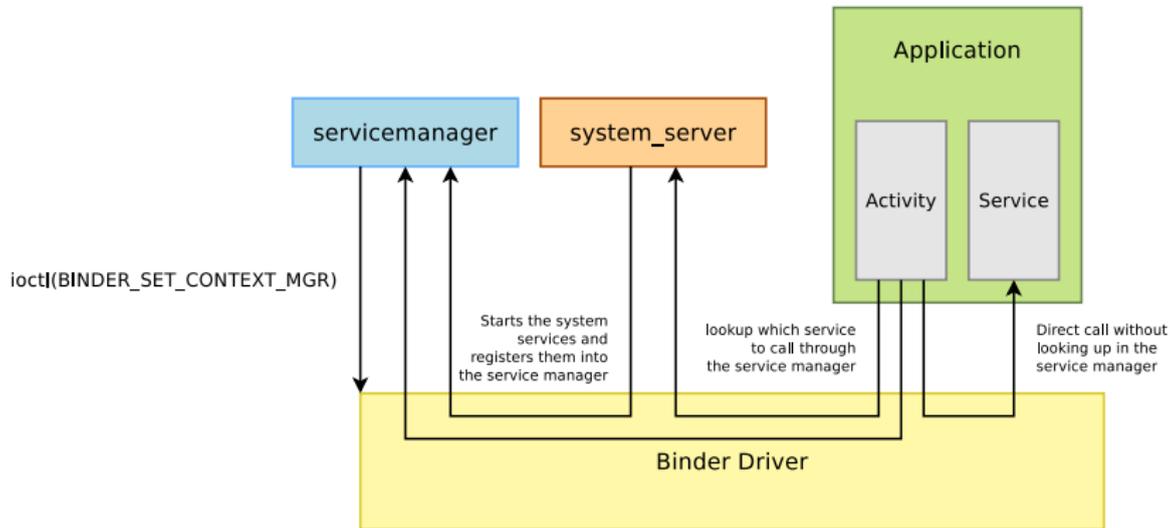
klogger

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 `printk()` from various parts of the kernel.
  - ▶ A syslog daemon, which handles the userspace 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

Logger

- 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

- 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

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

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

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 HTC G1, to allocate heaps used for 2D hardware acceleration
  - ▶ ADB
  - ▶ YAFFS2
  - ▶ Timed GPIOs

Android Bootloaders

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux
Developers
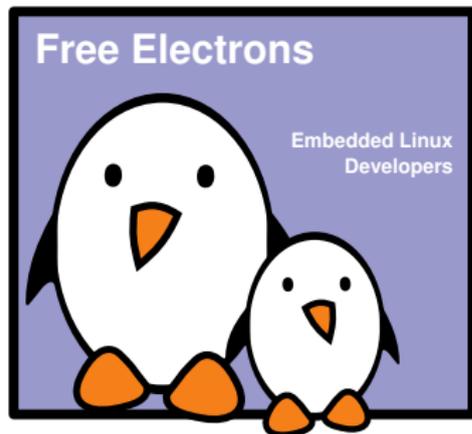
Boot Sequence

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 x86 (1)

- The x86 processors are typically bundled on a board with a non-volatile memory containing a program, the BIOS.
- This program gets executed by the CPU after reset, and is responsible for basic hardware initialization and loading of a small piece of code from non- volatile storage.
  - This piece of code is usually the first 512 bytes of a storage device
- This piece of code is usually a 1st stage bootloader, which will load the full bootloader itself.
- The bootloader can then offer all its features. It typically understands filesystem formats so that the kernel file can be loaded directly from a normal filesystem.

BIOS
from ROM

↓

Stage 1
512 bytes
from raw storage

↓

Stage 2
from raw storage

↓

Kernel
from filesystem
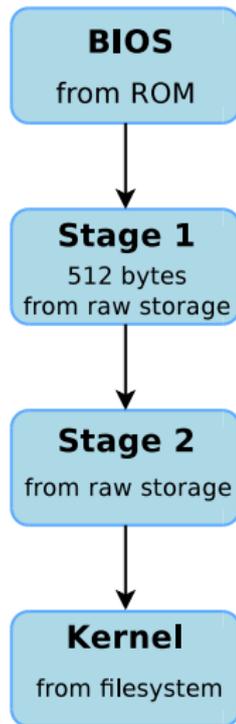
Bootloaders on x86 (2)

- 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:
    http://free-electrons.com/docs/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)

Physical memory

Execution starts → here

NOR

RAM
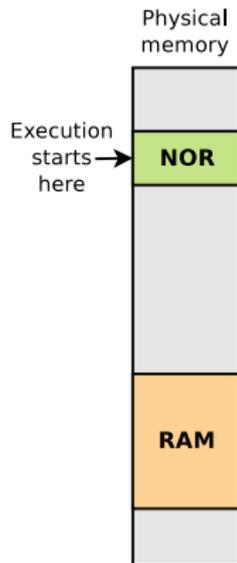
Booting on embedded CPUs: case 2

- ▶ 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, 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

RomBoot
stored in ROM
in the CPU

↓

AT91Bootstrap
stored in NAND or SPI flash
runs from SRAM

↓

U-Boot
stored in NAND or SPI flash
runs from DRAM

↓

Linux Kernel
stored in NAND, SD, network
runs from SDRAM
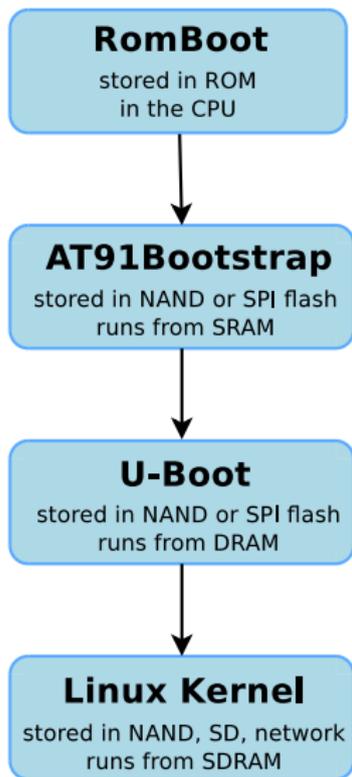
▶ **RomBoot**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to 4 KB. No user interaction possible in standard boot mode.

▶ **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 (bootloaders no longer exists).

ROM Code
stored in ROM
in the CPU

↓

X-Loader
stored in NAND or SD
runs from SRAM

↓

U-Boot
stored in NAND or SD
runs from DRAM

↓

Linux Kernel
stored in NAND, SD, network
runs from SDRAM
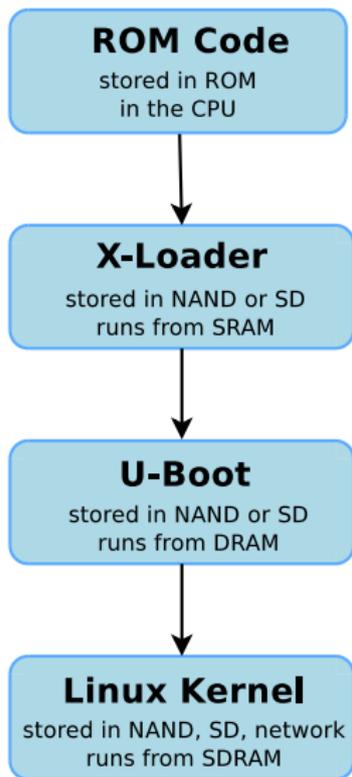
- ▶ **ROM Code**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM or RAM (RAM can be initialized by ROM code through a configuration header). Size limited to <64 KB. No user interaction possible.

- ▶ **X-Loader**: 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`.

- ▶ **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**, a new architecture-neutral bootloader, written as a successor of U-Boot. Better design, better code, active development, but doesn't yet have as much hardware support as U-Boot.
    http://www.barebox.org
- ▶ There are also a lot of other open-source or proprietary bootloaders, often architecture-specific
  - ▶ RedBoot, Yaboot, PMON, etc.

Fastboot

- Fastboot is a protocol to communicate 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
- Part of the Google's Test Suite, making it mandatory to obtain the Android Market

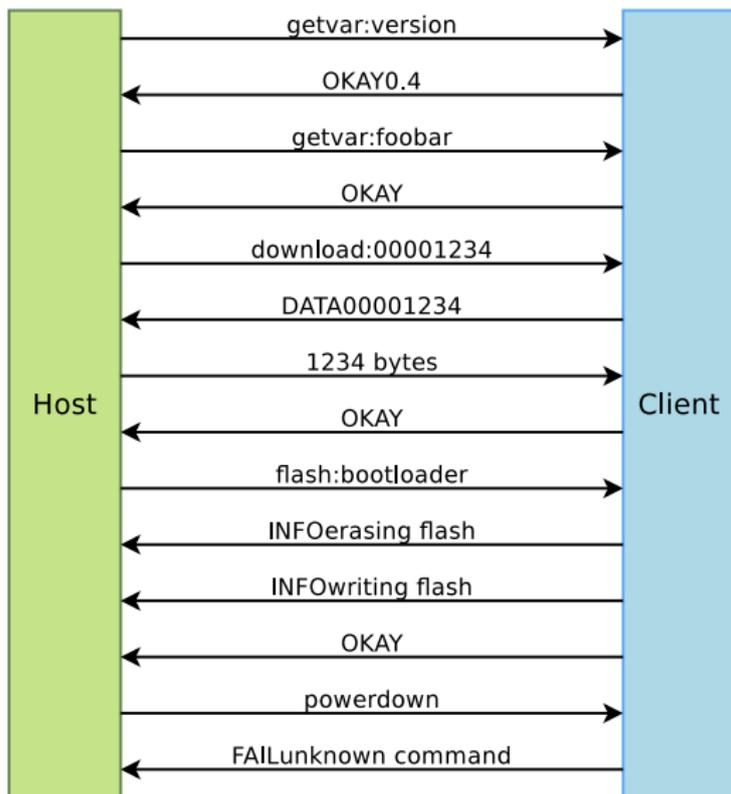# The Fastboot protocol

- It is very restricted, only 10 commands and 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

Commands available 1/2

- ▶ Vendor-specific commands must begin with an upper-case character. Commands beginning with a lower-case character are reserved for the Fastboot specifications and their evolution

- ▶ Commands defined by the Fastboot specifications are:

getvar:%s Reads a variable from the bootloader. It will be returned after the OKAY command.

download:%08x Writes data to memory on the device to be used later. The client either replies DATA\%08x if it succeeded or FAIL.

verify:%08x Sends a digital signature to verify the previously downloaded data. Required if the device is secure.

flash:%s Writes the previously sent data to the given partition.

erase:%s   Erases the given flash partition (sets all the partition to `0xFF`

boot   Means that what was last downloaded is a root filesystem, and instructs to boot on it.

continue   Orders the device to continue booting as usual

reboot   Reboots the device

reboot-bootloader   Reboots back in the bootloader mode

powerdown   Powers off the device

Defined 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.

▶ They are retrieved through the `getvar` command.

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?

Developing and Debugging with ADB

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux
Developers

Introduction

ADB
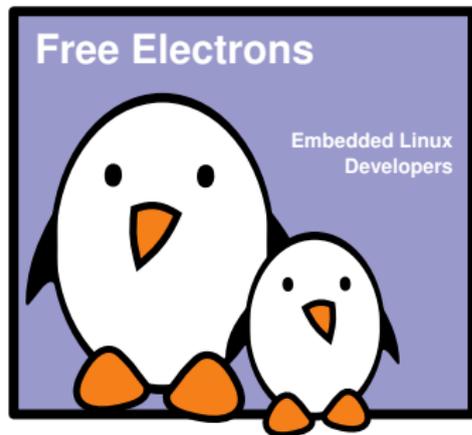
- ▶ Usually on embedded devices, debugging and is done either through a serial port on the device or JTAG for low-level debugging
- ▶ This setup works well when developing a new product that will have a static system. You develop and debug a system on a product with serial and JTAG ports, and remove 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
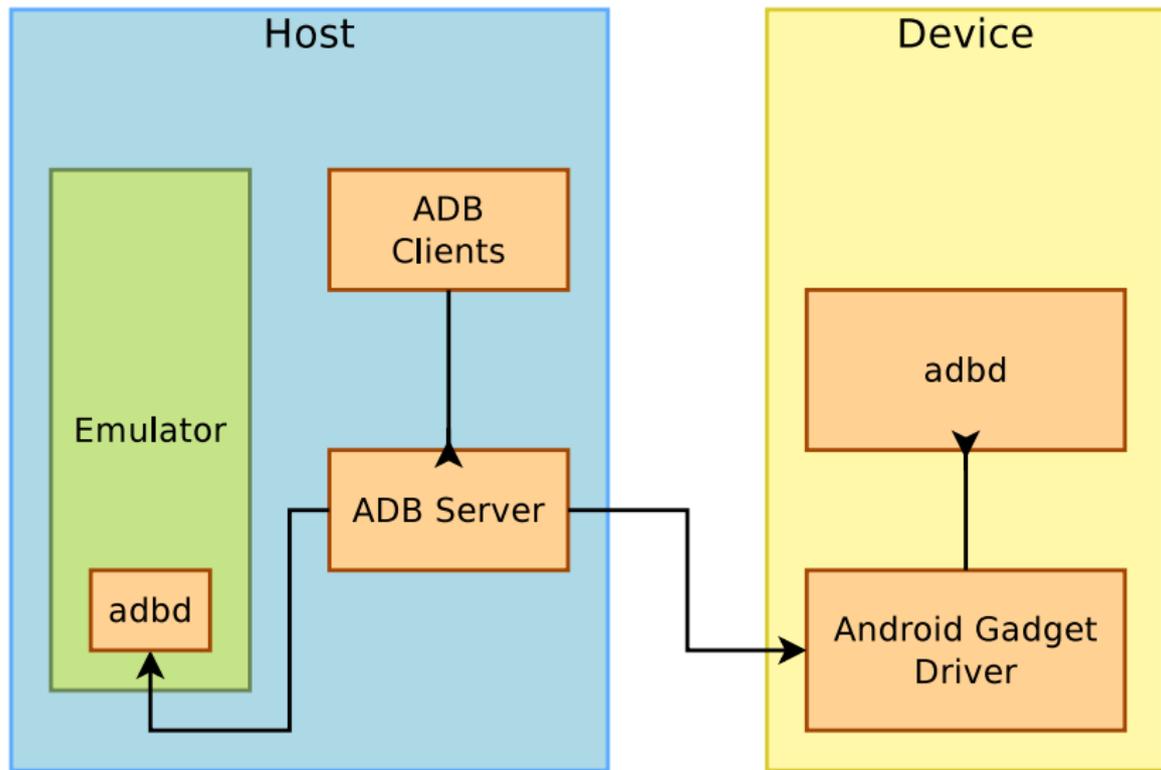
Use of ADB

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
  ▶ Only if the `ro.secure` property is to 1 to force ADB into user mode, and `ro.debuggable` is 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

lolcat Alias to `adb logcat`

hell Equivalent to `adb shell`, with a different color scheme

Examples
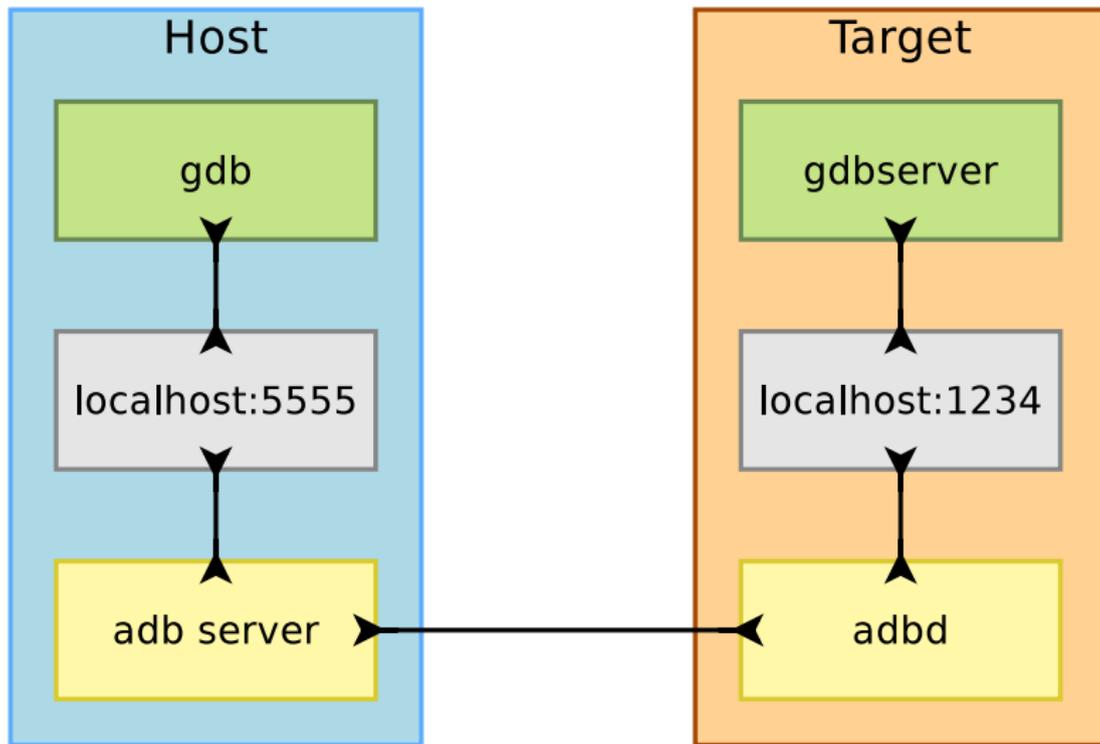
```
adb forward tcp:5555 tcp:1234
```
See also `gdbclient`
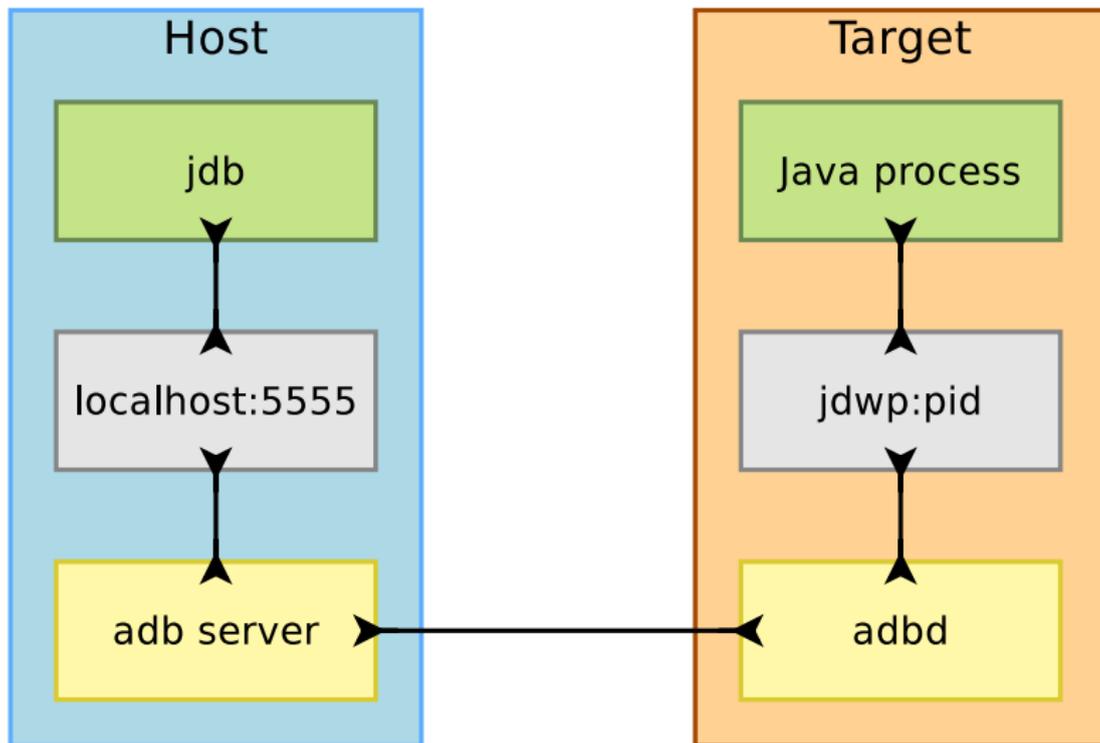
```
adb forward tcp:5555 jwdp:4242
```

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.free-electrons.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`

Android Filesystem

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux
Developers
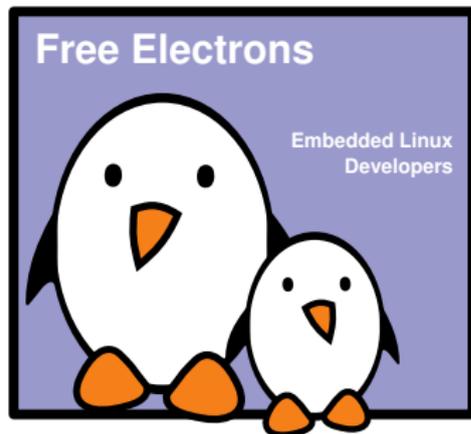
Principle and solutions

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 performances. umount makes sure that those writes are committed to the storage.

- 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 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 a 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 a NAND Flash
  - `root=/dev/mtdblockX`, where `X` is the partition number
  - `/dev/mtdblock3` is the fourth partition of the NAND flash (if only one NAND flash is present)

Mounting rootfs over the network (1)

Once networking works, your root filesystem could be a directory on your GNU/Linux development host, exported by NFS (Network File System). This is very convenient for system development:

- ▶ Makes it very easy to update files on the root filesystem, without rebooting. Much faster than through the serial port.
- ▶ Can have a big root filesystem even if you don't have support for internal or external storage yet.
- ▶ The root filesystem can be huge. You can even build native compiler tools and build all the tools you need on the target itself (better to cross-compile though).
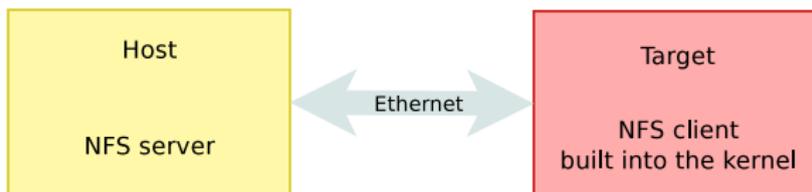
Mounting rootfs over the network (2)

On the development workstation side, a NFS server is needed

- ▶ Install an NFS server (example: Debian, Ubuntu)
  `sudo apt-get install nfs-kernel-server`
- ▶ Add the exported directory to your /etc/exports file:
  `/home/tux/rootfs 192.168.1.111(rw,no_root_squash,`
  `no_subtree_check)`
  - ▶ 192.168.1.111 is the client IP address
  - ▶ `rw,no_root_squash,no_subtree_check` are the NFS server
    options for this directory export.
- ▶ Start or restart your NFS server (example: Debian, Ubuntu)
  `sudo /etc/init.d/nfs-kernel-server restart`

Mounting rootfs over the network (3)

- On the target system
- The kernel must be compiled with
    - `CONFIG_NFS_FS=y` (NFS support)
    - `CONFIG_IP_PNP=y` (configure IP at boot time)
    - `CONFIG_ROOT_NFS=y` (support for NFS as rootfs)
- The kernel must be booted with the following parameters:
    - `root=/dev/nfs` (we want rootfs over NFS)
    - `ip=192.168.1.111` (target IP address)
    - `nfsroot=192.168.1.110:/home/tux/rootfs/` (NFS server details)
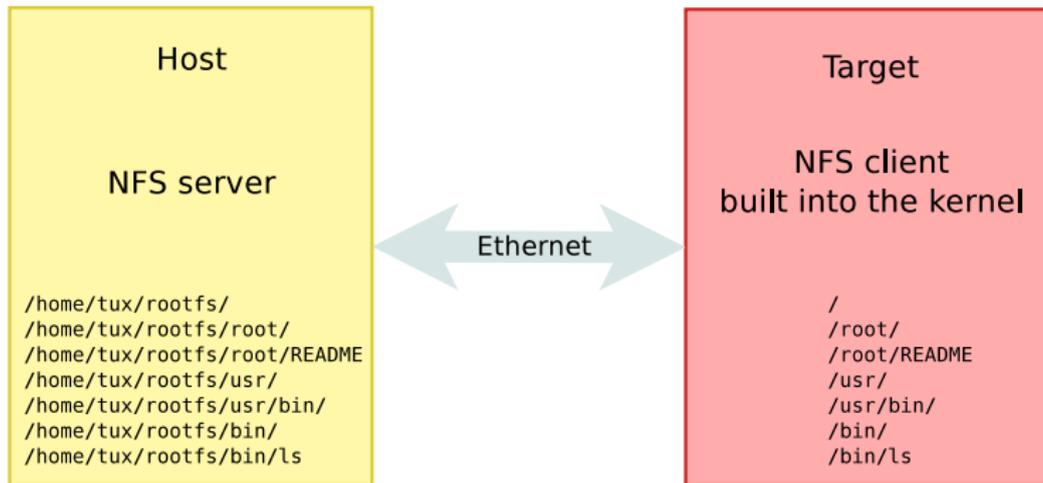
rootfs in memory: initramfs (1)

- 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
- 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 (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
- ▶ `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

/system

./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), therefore avoiding collisions. /sys)
- ▶ There is some collision 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 role is to **allow applications to access hardware devices**
- In the Linux kernel, most devices are presented to userspace 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

Devices: everything is a file

- A very important Unix design decision was to represent most of the "system objects" as files
- It allows applications to manipulate all "system objects" with the normal file API (`open`, `read`, `write`, `close`, etc.)
- So, devices had to be represented as files to the applications
- This is done through a special artifact called a **device file**
- It is a special type of file, that associates a file name visible to userspace applications to the triplet *(type, major, minor)* that the kernel understands
- All *device files* are by convention stored in the `/dev` directory

Device files examples

Example of device files in a Linux system

```
$ ls -l /dev/ttyS0 /dev/tty1 /dev/sda1 /dev/sda2 /dev/zero
brw-rw---- 1 root disk    8,  1 2011-05-27 08:56 /dev/sda1
brw-rw---- 1 root disk    8,  2 2011-05-27 08:56 /dev/sda2
crw------- 1 root root    4,  1 2011-05-27 08:57 /dev/tty1
crw-rw---- 1 root dialout 4, 64 2011-05-27 08:56 /dev/ttyS0
crw-rw-rw- 1 root root    1,  5 2011-05-27 08:56 /dev/zero
```

Example C code that uses the usual file API to write data to a
serial port

```
int fd;
fd = open("/dev/ttyS0", O_RDWR);
write(fd, "Hello", 5);
close(fd);
```

Creating device files

- ▶ On a basic Linux system, the device files have to be created manually using the mknod command
  - ▶ `mknod /dev/<device> [c|b] major minor`
  - ▶ Needs root privileges
  - ▶ Coherency between device files and devices handled by the kernel is left to the system developer
- ▶ On more elaborate Linux systems, mechanisms can be added to create/remove them automatically when devices appear and disappear
  - ▶ `devtmpfs` virtual filesystem, since kernel 2.6.32
  - ▶ `udev` daemon, solution used by desktop and server Linux systems
  - ▶ `mdev` program, a lighter solution than `udev`

Virtual Filesystems

proc virtual filesystem

▶ The `proc` virtual filesystem exists since the beginning of Linux
▶ It allows
  ▶ The kernel to expose statistics about running processes in the system
  ▶ The user to adjust at runtime various system parameters about process management, memory management, etc.
▶ The `proc` filesystem is used by many standard userspace applications, and they expect it to be mounted in /proc
▶ Applications such as `ps` or `top` would not work without the `proc` filesystem
▶ Command to mount /proc:
  `mount -t proc nodev /proc`
▶ `Documentation/filesystems/proc.txt` in the kernel sources
▶ `man proc`

proc contents

- One directory for each running process in the system
  - /proc/<pid>
  - cat /proc/3840/cmdline
  - It contains details about the files opened by the process, the CPU and memory usage, etc.
- /proc/interrupts, /proc/devices, /proc/iomem, /proc/ioports contain general device-related information
- /proc/cmdline contains the kernel command line
- /proc/sys contains many files that can be written to to adjust kernel parameters
  - They are called *sysctl*. See Documentation/sysctl/ in kernel sources.
  - Example
    echo 3 > /proc/sys/vm/drop_caches

sysfs filesystem

- ▶ The `sysfs` filesystem is a feature integrated in the 2.6 Linux kernel
- ▶ It allows to represent in userspace the vision that the kernel has of the buses, devices and drivers in the system
- ▶ It is useful for various userspace applications that need to list and query the available hardware, for example udev or mdev
- ▶ All applications using sysfs expect it to be mounted in the `/sys` directory
- ▶ Command to mount `/sys`:
  `mount -t sysfs nodev /sys`
- ▶ $ ls /sys/
  block bus class dev devices firmware
  fs kernel modulepower

Minimal filesystem

Basic applications

- In order to work, a Linux system needs at least a few applications
- An `init` application, which is the first userspace application started by the kernel after mounting the root filesystem
  - The kernel tries to run `/sbin/init`, `/bin/init`, `/etc/init` and `/bin/sh`.
  - If none of them are found, the kernel panics and the boot process is stopped.
  - The init application is responsible for starting all other userspace applications and services
- Usually a shell, to allow a user to interact with the system
- Basic Unix applications, to copy files, move files, list files (commands like `mv`, `cp`, `mkdir`, `cat`, etc.)
- Those basic components have to be integrated into the root filesystem to make it usable

Bootloader
loads the kernel to RAM and starts it

↓

Kernel
initializes hardware devices and kernel subsystems
mounts the root filesystem indicated by root=
starts the init application, /sbin/init by default

↓

/sbin/init
starts other userspace services and applications

Shell

Other application

Root filesystem
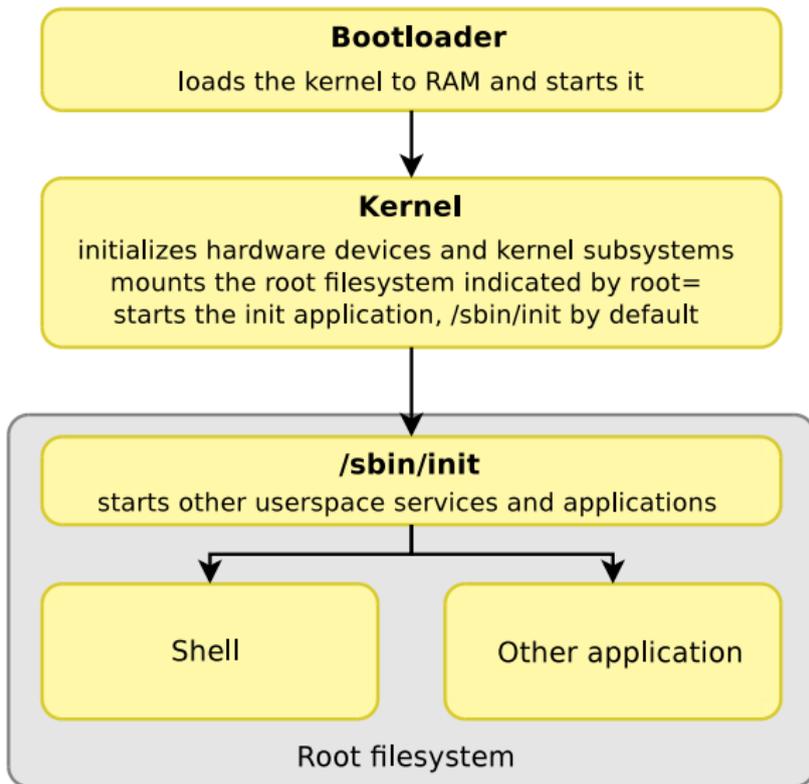
Android Build System

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux
Developers
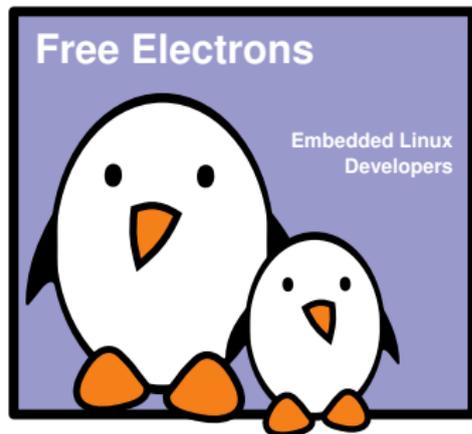
Basics

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
```

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
- ▶ It also generates the system images in the `out/target/product/<product_name>/` directory

Cleaning

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

envsetup.sh

Purpose

- Obviously modifies the current environment, that's why we have to `source` it
- Exports some environment variables that will be used across the build system to customize its behaviour
- Adds some useful macros as well, making Android easier to compile, to deal with this huge source tree, to search for patterns across the source code, etc.

Environments variables 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 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

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

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

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 either `generic` or `full`, which contains more apps and more locales
- `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_SIMULATOR`
    - Are we building the simulator? (different from the *emulator*)
- `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.

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.

```
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
- `LOCAL_PACKAGE_NAME`
    - Equivalent to `LOCAL_MODULE` for Android packages
- `LOCAL_C_INCLUDES`
    - List of paths to extra headers used by this module
- `LOCAL_PRELINK_MODULES`
    - Does the module uses library pre-linking? (if you set this to `true`, you will have to modify `build/core/prelink-linux-arm.map` 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

Make and clean a module

- To build a module from the top directory, just do
  `make ModuleName`
- The files generated will be put in
  `out/target/product/<product_name>/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.
- To clean a single module, do `make clean-ModuleName`
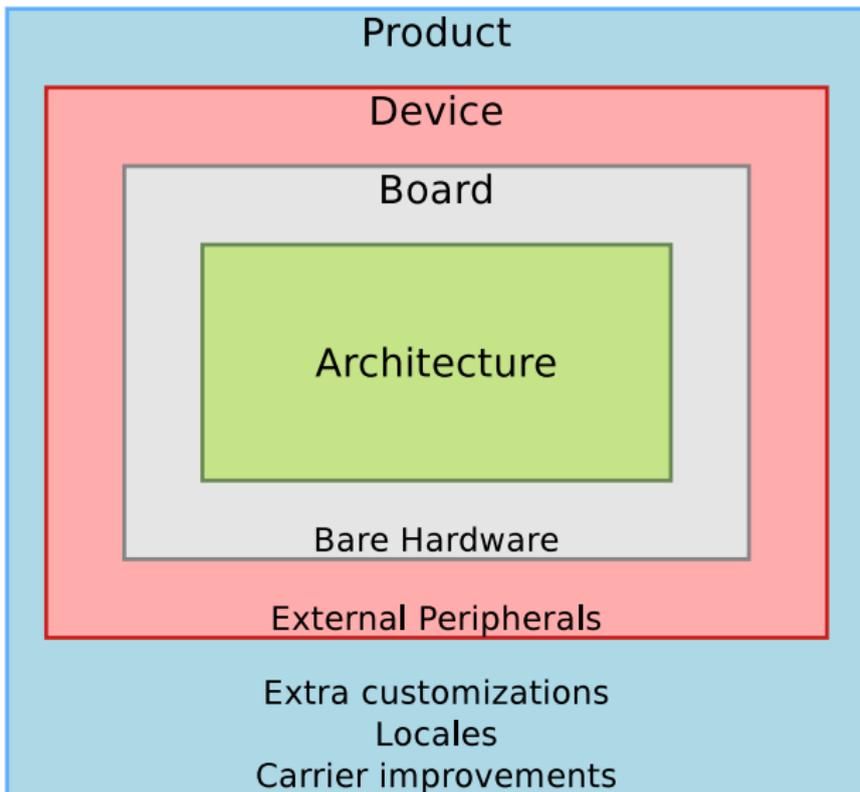
Add a New Product

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

Device

Board

Architecture

Bare Hardware

External Peripherals

Extra customizations
Locales
Carrier improvements

```
$(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
```

```
$(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` variable 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

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

PRODUCT_PACKAGES += FooBar

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

DEVICE_PRODUCT_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
- You can have a full working example in `device/samsung/crespo/BoardConfigCommon.mk`
- 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

```
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 Cortex-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 as filesystem for our generated partitions
- ► BOARD_SYSTEMIMAGE_PARTITION_SIZE
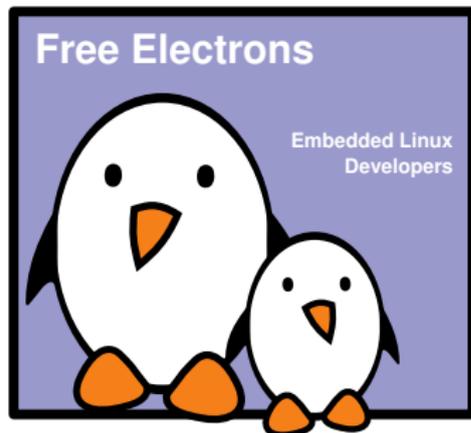 - ► Size of the system partitions in bytes.

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

Android Native Layer

Maxime Ripard
Free Electrons

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.
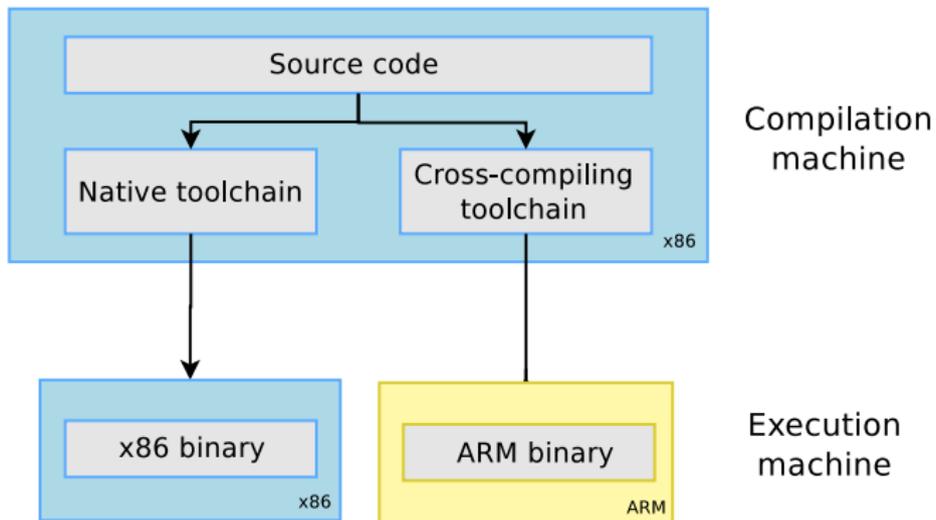
Machines in build procedures

- 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.
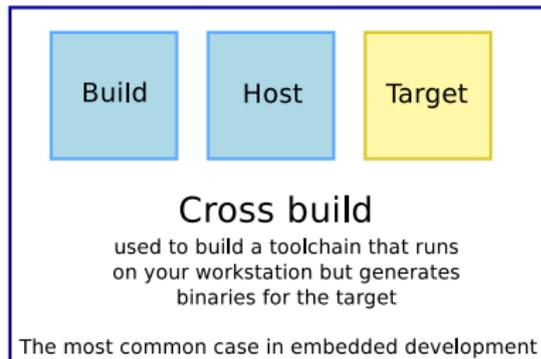- Four common build types are possible for toolchains

Native build
used to build the normal gcc
of a workstation

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

The most common case in embedded development

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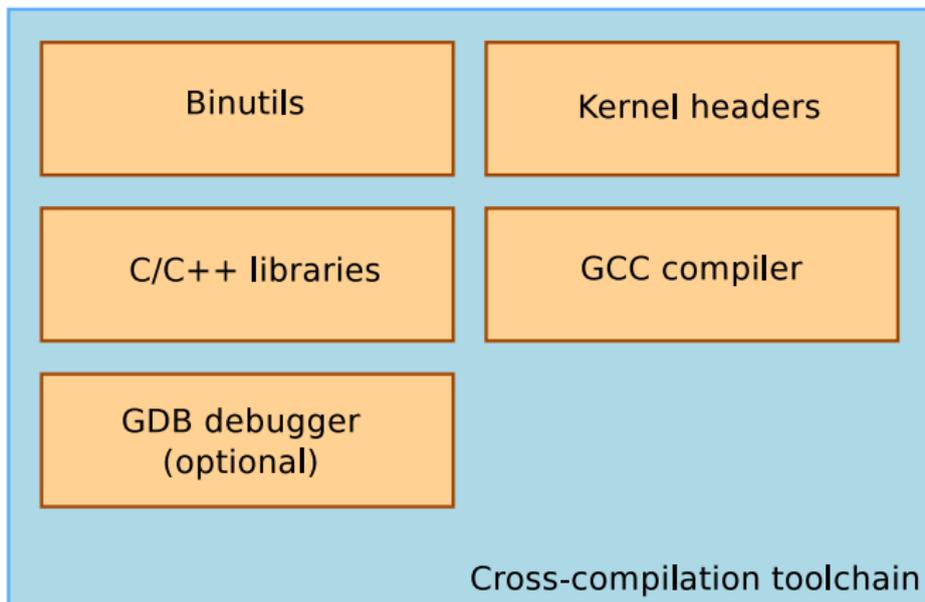used to build a toolchain that runs on your
target and generates binaries for the target

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

Binutils

Kernel headers

C/C++ libraries

GCC compiler

GDB debugger
(optional)

Cross-compilation toolchain

Binutils

- **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 !
  - `strip`, to strip useless parts of binaries in order to reduce 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
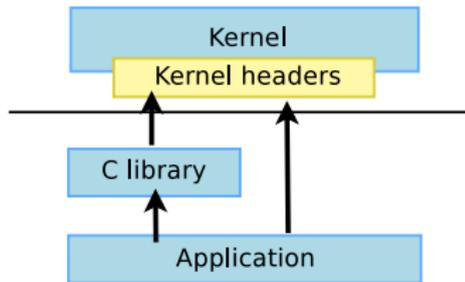
Kernel headers (2)

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

  ```
  #define __NR_ext          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-userspace 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.

GCC compiler

- GNU C Compiler, the famous free software compiler
- Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, and generate code for a large number of CPU architectures, including ARM, AVR, Blackfin, CRIS, FRV, M32, MIPS, MN10300, PowerPC, SH, v850, i386, x86_64, IA64, Xtensa, etc.
- http://gcc.gnu.org/
- Available under the GPL license, libraries under the LGPL.

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*, *eglibc*, *dietlibc*, *newlib*, etc.
- ▶ The choice of the C library must be made at the time of the cross-compiling toolchain generation, as the GCC compiler is compiled against a specific C library.

Kernel

C library

Application
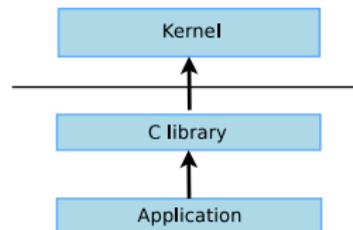
Bionic

Bionic 1/2

▶ 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 `getprotobyname()`

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 userspace. 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://j.mp/L53Tuu for details about Bionic.

Toolbox

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, those 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, bbsh,
brctl, bunzip2, busybox, bzcat, bzip2, cal, cat, catv, chat, chattr, chcon, chgrp, chmod,
chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm, cp, cpio, crond, crontab,
cryptpw, cttyhack, cut, date, dc, dd, deallocvt, delgroup, deluser, depmod, devfsd, df,
dhcprelay, diff, dirname, dmesg, dnsd, dos2unix, dpkg, dpkg_deb, du, dumpkmap, dumpleases,
e2fsck, echo, ed, egrep, eject, env, envdir, envuidgid, ether_wake, expand, expr, fakeidentd,
false, fbset, fbsplash, fdflush, fdformat, fdisk, fetchmail, fgrep, find, findfs, fold, free,
freeramdisk, fsck, fsck_minix, ftpget, ftpput, fuser, getenforce, getopt, getsebool, getty,
grep, gunzip, gzip, halt, hd, hdparm, head, hexdump, hostid, hostname, httpd, hush, hwclock,
id, ifconfig, ifdown, ifenslave, ifup, inetd, init, inotifyd, insmod, install, ip, ipaddr,
ipcalc, ipcrm, ipcs, iplink, iproute, iprule, iptunnel, kbd_mode, kill, killall, killall5,
klogd, lash, last, length, less, linux32, linux64, linuxrc, ln, load_policy, loadfont,
loadkmap, logger, login, logname, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lzmacat,
makedevs, man, matchpathcon, md5sum, mdev, mesg, microcom, mkdir, mke2fs, mkfifo, mkfs_minix,
mknod, mkswap, mktemp, modprobe, more, mount, mountpoint, msh, mt, mv, nameif, 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, sulogin, sum, sv, svlogd, swapoff, swapon, switch_root, sync, sysctl,
syslogd, tac, tail, tar, taskset, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time, top,
touch, tr, traceroute, true, tty, ttysize, tune2fs, udhcpc, udhcpd, udpsvd, umount, uname,
uncompress, unexpand, uniq, unix2dos, unlzma, unzip, uptime, usleep, uudecode, uuencode,
vconfig, vi, vlock, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat, zcip

Toolbox

- ▶ 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 Gingerbread

alarm, cat, chmod, chown, cmp, date, dd, df, dmesg, exists, getevent, getprop, hd, id,
ifconfig, iftop, insmod, ioctl, ionice, kill, ln, log, ls, lsmod, lsof, mkdir, mount, mv,
nandread, netstat, newfs_msdos, notify, powerd, printenv, ps, r, readtty, reboot, renice, rm,
rmdir, rmmod, rotatefb, route, schedtop, sendevent, setconsole, setkey, setprop, sleep, smd,
start, stop, sync, syren, top, umount, uptime, vmstat, watchprops, wipe

Init

Init

- `init` is the name of the first userspace 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 hotpluggingm, 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, `init.rc` itself, and `init.<platform_name>.rc`
- ▶ While `init.rc` is always taken into account, `init.<platform_name>.rc` is only interpreted if the platform currently running the system reports the same name
- ▶ 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

```
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

    write /proc/cpu/alignment 4
```

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.

```
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

ueventd.rc syntax

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

- Example

```
/dev/bus/usb/*           0660    root        usb
```

Various daemons

Vold

- ▶ 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

- `rild` is the Radio Interface Layer Daemon
- This daemon drives the telephony stack, both voice and data communications
- 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.
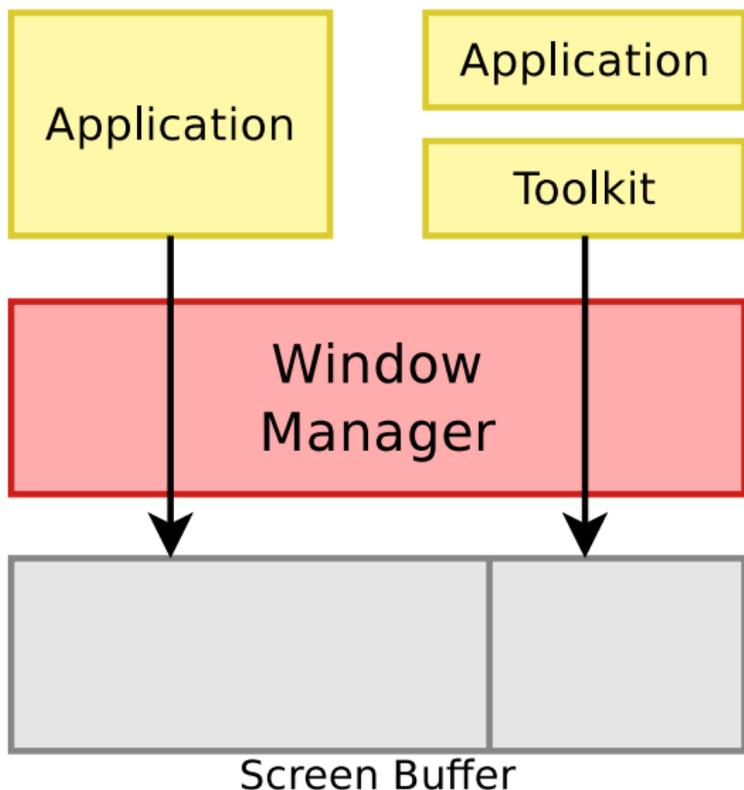   - *Unsolicited commands*: commands that come from the baseband: receiving an SMS, a call, signal strength changed, etc.

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.
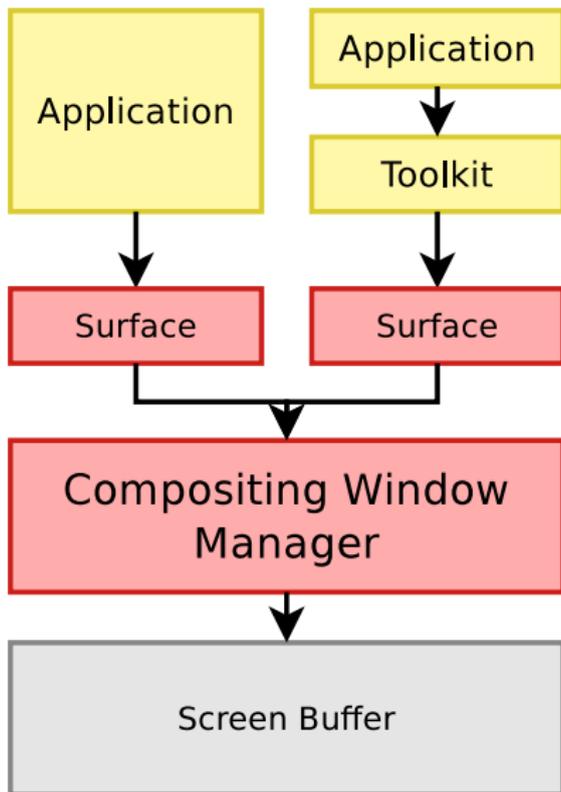
SurfaceFlinger and PixelFlinger

Application

Application
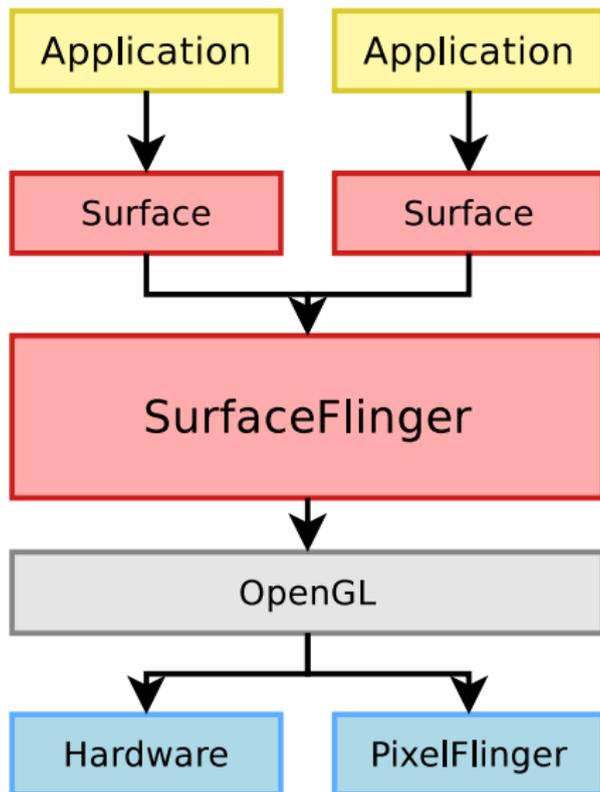
Toolkit

Window Manager

Screen Buffer

SurfaceFlinger

- 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.
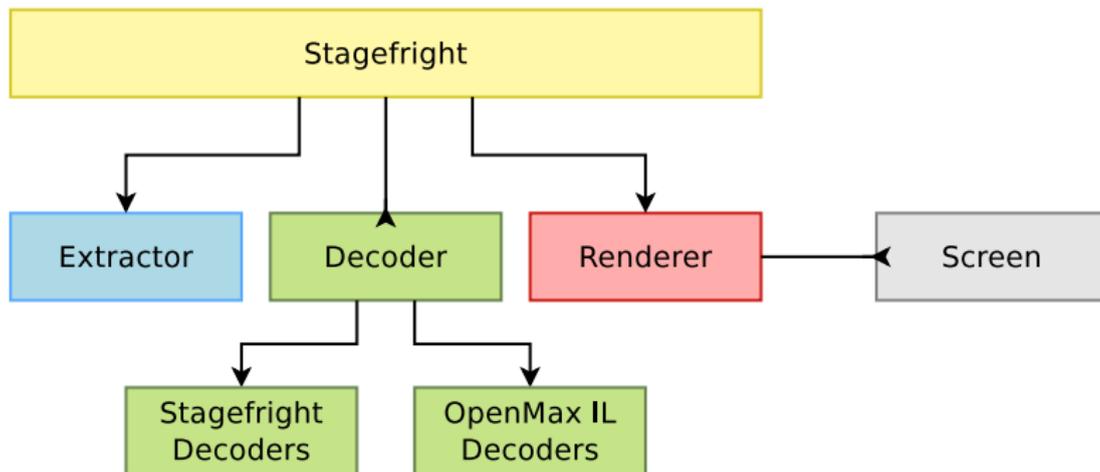
Stagefright

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 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 `OMXCodec.cpp` file, in the `kDecoderInfo` array.
    - → No runtime extension of the decoders, this is done at compilation time.

```
static const CodecInfo kDecoderInfo[] = {
    { MEDIA_MIMETYPE_AUDIO_AAC, "OMX.TI.AAC.decode" },
    { MEDIA_MIMETYPE_AUDIO_AAC, "AACDecoder" },
};
```

Dalvik and Zygote

Dalvik

- ▶ 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

Zygote

- 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

Hardware Abstraction Layers

- Usually, the kernel already provides a HAL for userspace
- 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 userspace interface differs
  - All the code in the kernel must be GPL-licensed
- Google implemented its HAL with dynamically loaded userspace 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_legacy/include/hardware_legacy/AudioHardwareInterface.h`
- Graphics (`gralloc.so`, `copybit.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/copybit.h`
- Camera (`libcamera.so`) handles the camera functions: autofocus, take a picture, etc.
 - `frameworks/base/include/camera/CameraHardwareInterface.h`

Various layers

- GPS (`libgps.so`) configuration, data acquisition
  - `hardware/libhardware/include/hardware/gps.h`
- 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 the communications 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`
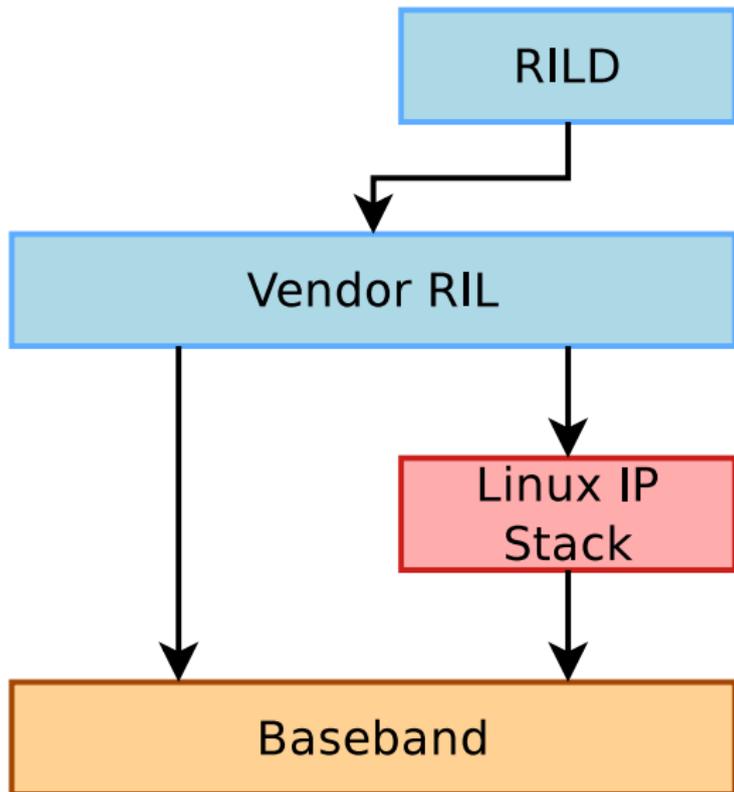
JNI

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 userspace 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

```
#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
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

```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);
}
```

```
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;
}
```
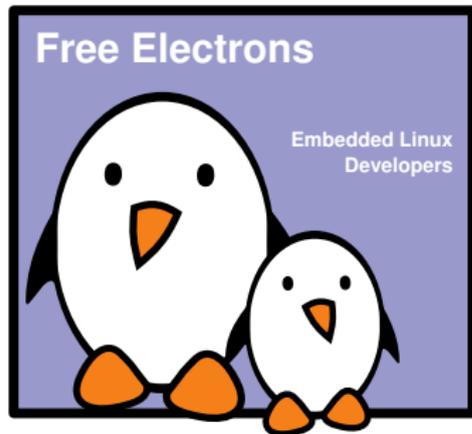
Android Framework and Applications

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux
Developers

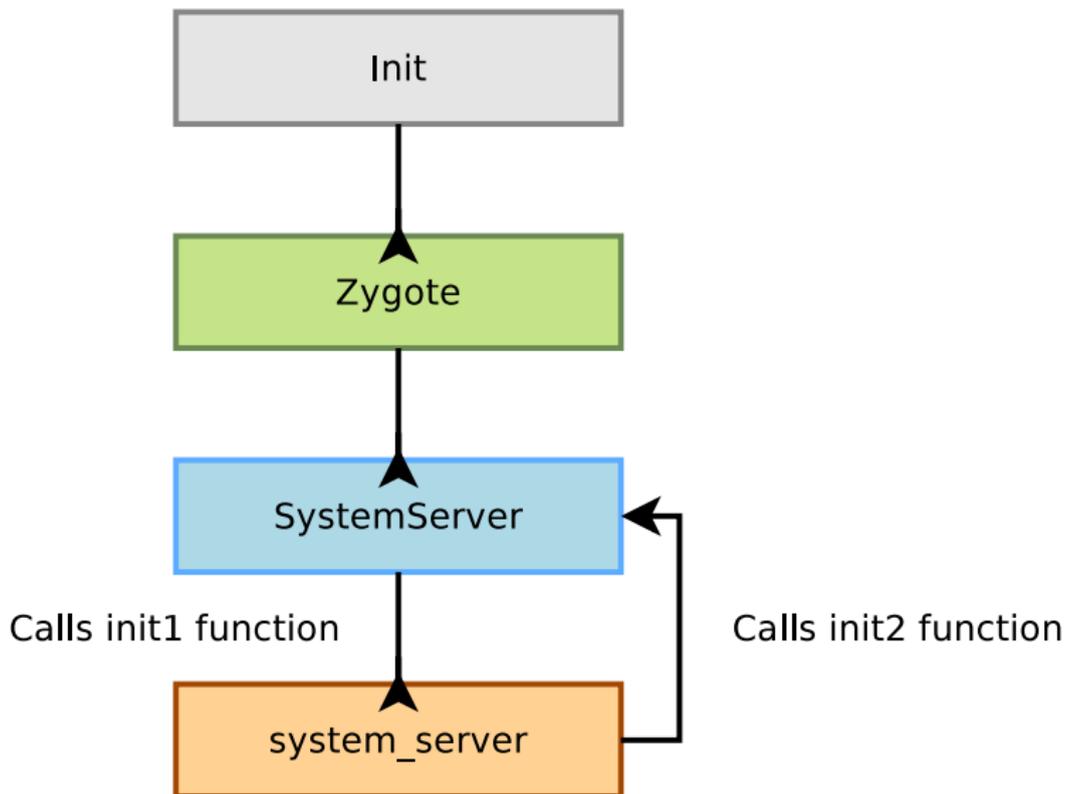# Service Manager and Various Services
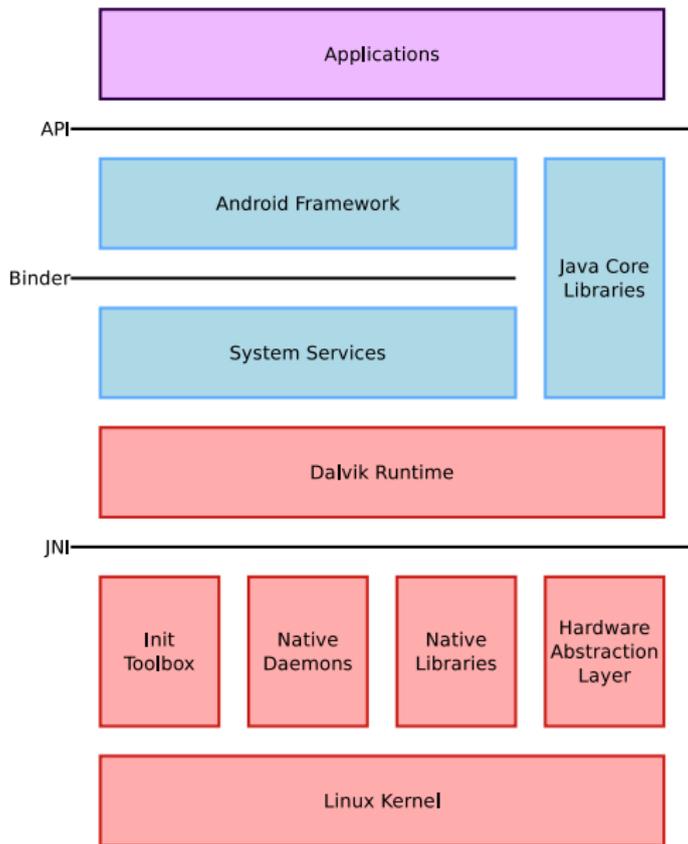
The first step: `system_server.c`

- Located in `frameworks/base/cmds/system_server`
- Started by `Zygote` through the SystemServer
- Starts all the various native services:
    - `SurfaceFlinger`
    - `SensorService`
    - `AudioFlinger`
    - `MediaPlayerService`
    - `CameraService`
    - `AudioPolicyService`
- 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 Communications, Binder and AIDLs

IPCs

- ▶ 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, we have several IPC mechanisms 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.

Binder 2/2

- 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 terminology

- *The Binder*
  - The overall Binder Architecture
- *Binder Interface*
  - A well-defined set of methods and properties other can call, and that should be implemented by *a* binder
- *A Binder*
  - A particular implementation of a Binder interface
- *Binder Object*
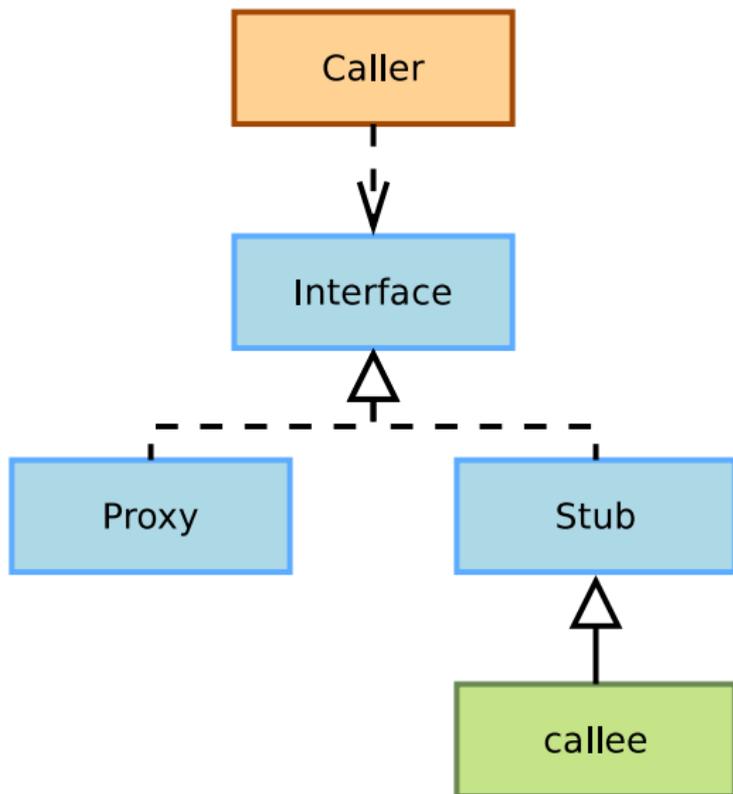 - An instance of a class that implements a Binder interface
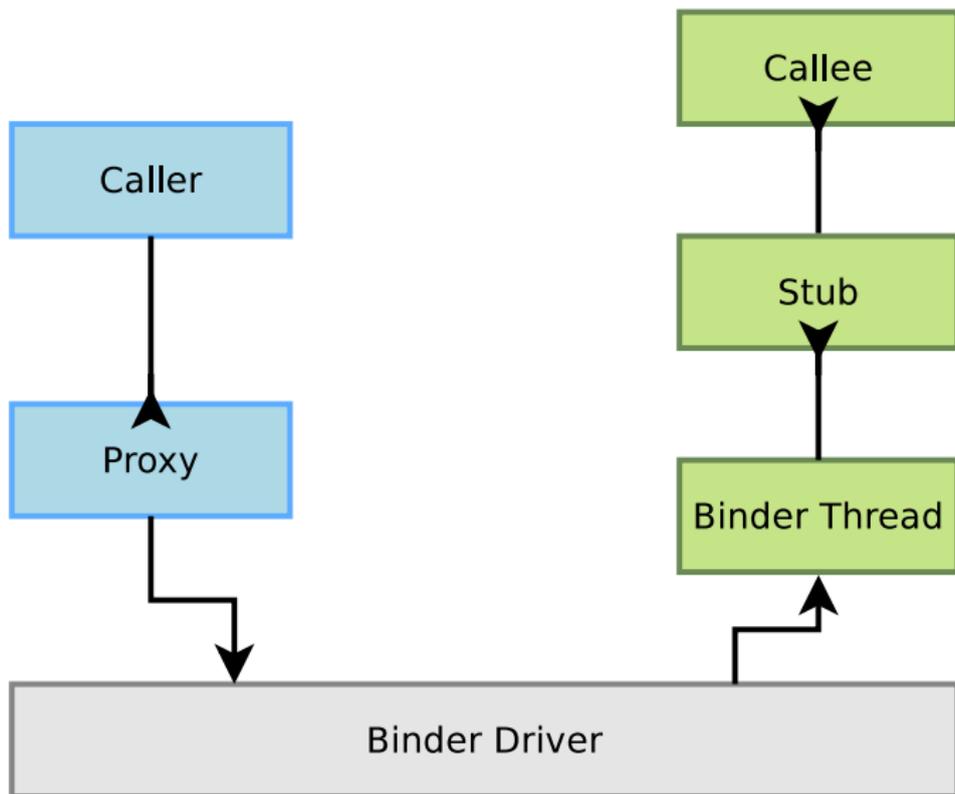
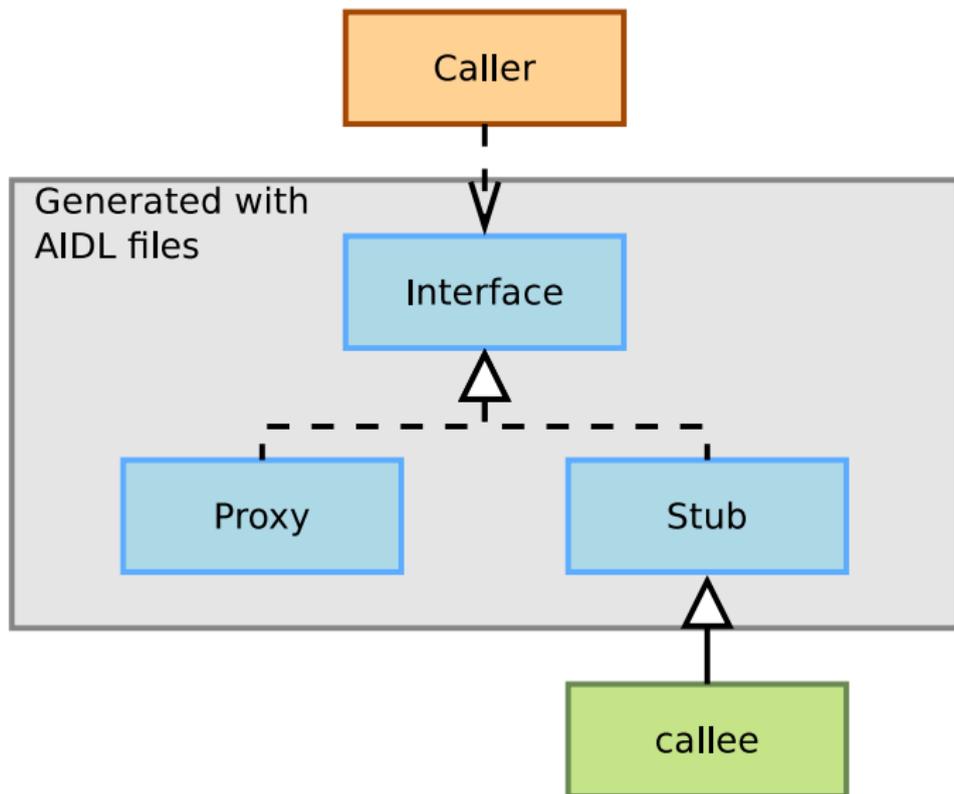# Android Interface Definition Language (AIDL)

- ▶ 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`

```
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 best target for the Intent by itself, possibly asking the user what to do if there are several matches

Various Java Services

Android Java Services

- 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

PowerManager

- ▶ 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
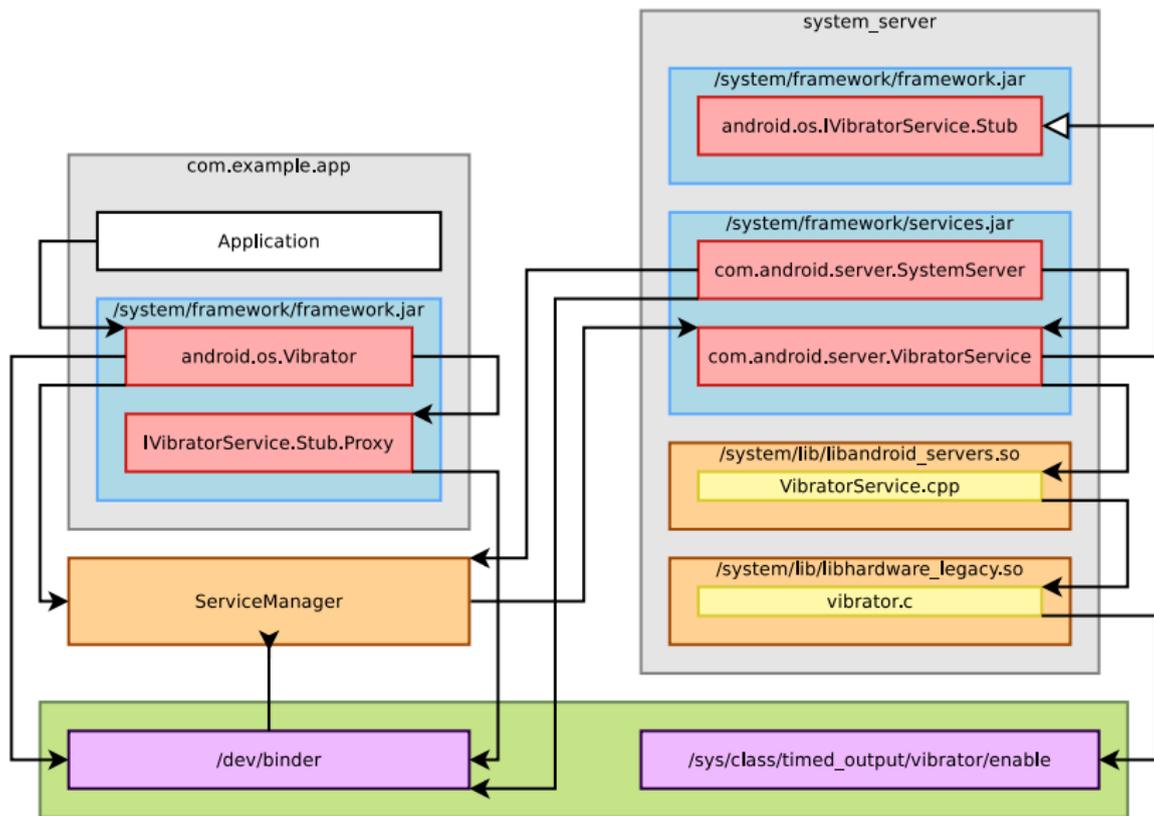
Extend the framework

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`

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-java-files)

LOCAL_MODULE_TAGS := optional

LOCAL_MODULE:= com.example.android.platform_library

include $(BUILD_JAVA_LIBRARY)
```

PlatformLibrary permissions file

```xml
<?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)
```
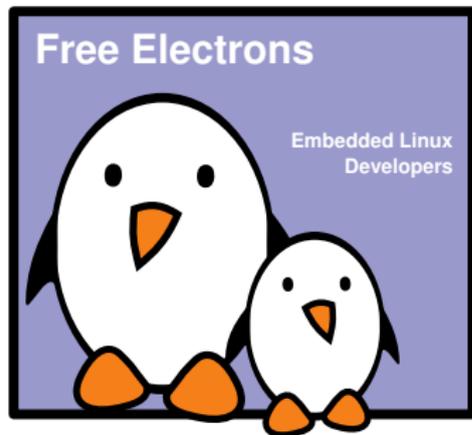
Android Application Development

Maxime Ripard
Free Electrons

Free Electrons

Embedded Linux
Developers

Basics

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 is 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

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.example.android">
    <application>
        <activity android:name=".ExampleActivity"
                  android:label="@string/example_label">
        </activity>
    </application>
</manifest>
```

NDK

- ▶ 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
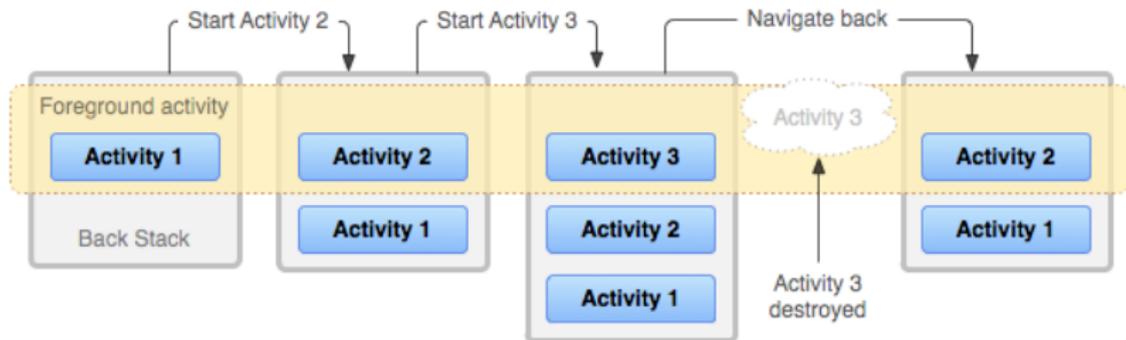
Activities

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
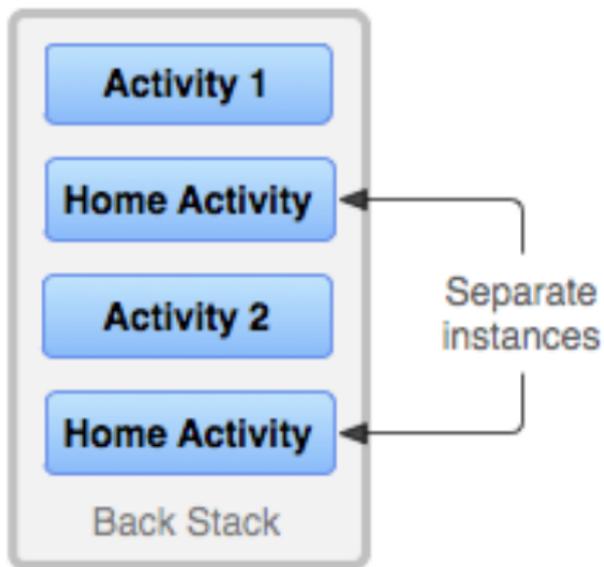
Credits: http://developer.android.com

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

- 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.

Credits: http://developer.android.com
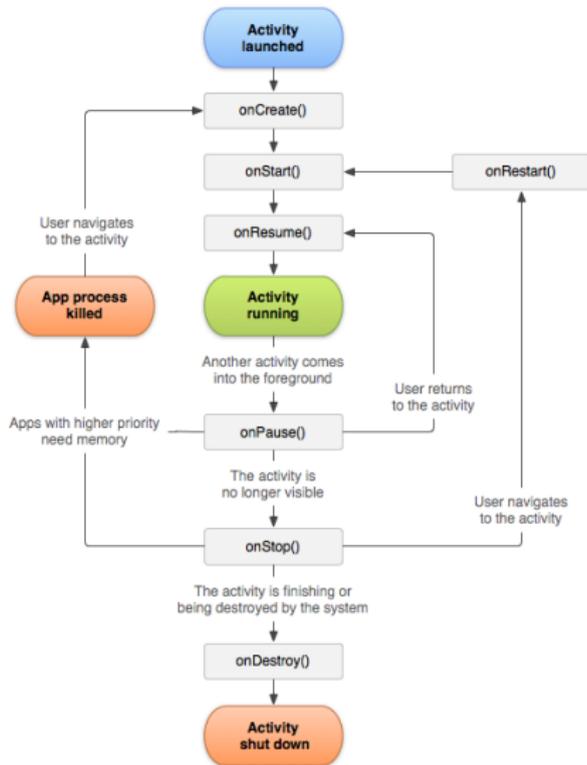
- As applications tend to be killed and restarted quite often, we need a way to store our internal state when killed and reload it when restarted
- Once again, this is done through callbacks
- Before killing the application, the system calls the onSaveInstanceState callback and when restarting it, it 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 make 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
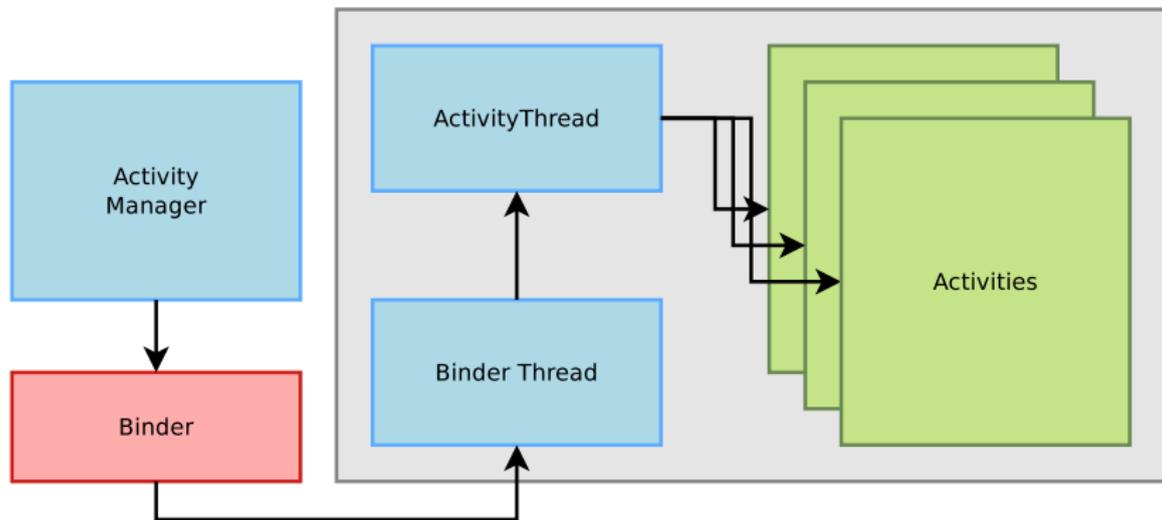
Credits: http://developer.android.com

Credits: `http://developer.android.com`
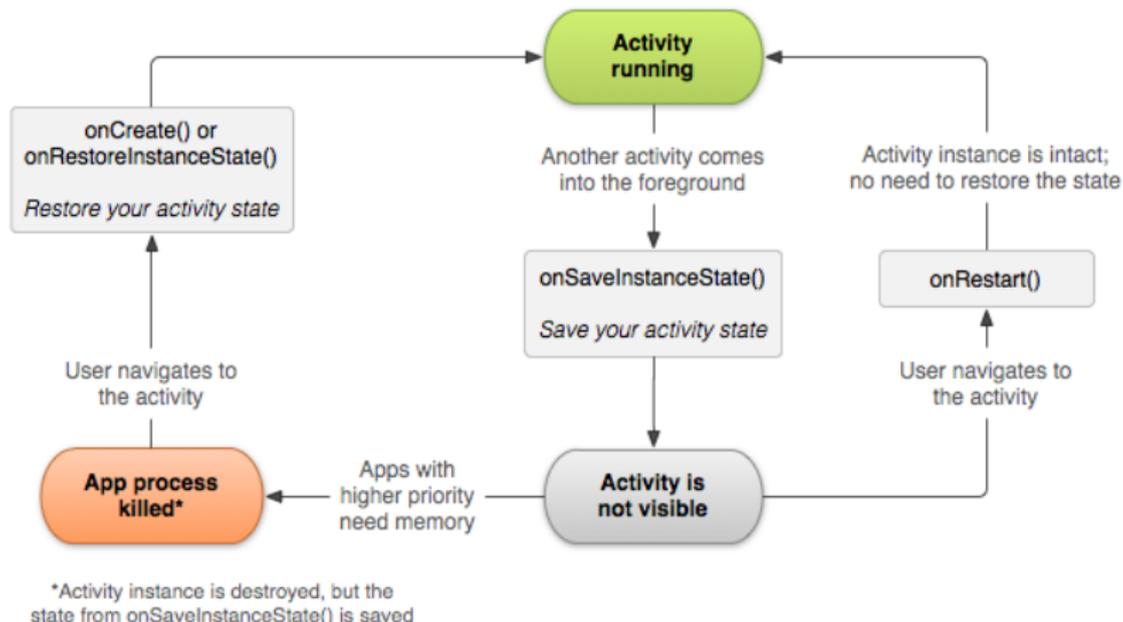
```java
public class ExampleActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.i("ExampleActivity", "Activity created!");
    }
    protected void onStart() {
        super.onStart();
    }
    protected void onResume() {
        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

```xml
<?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

Credits: `http://developer.android.com`
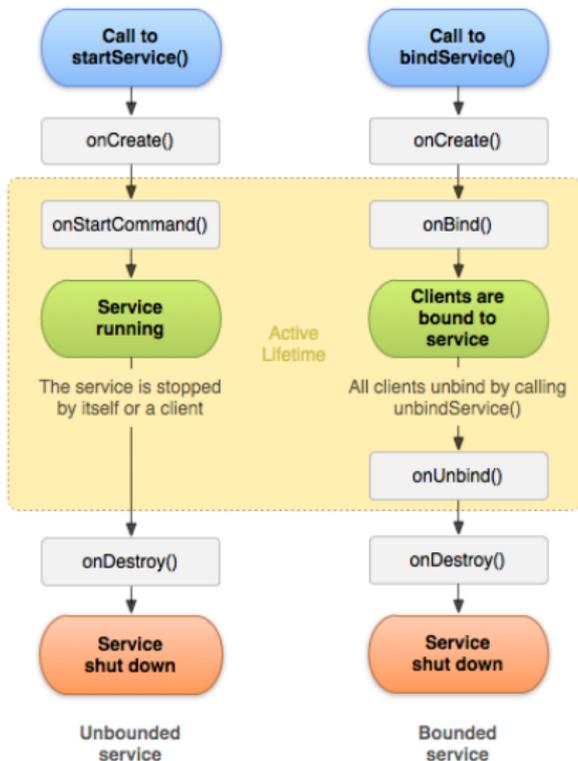
Bound Services
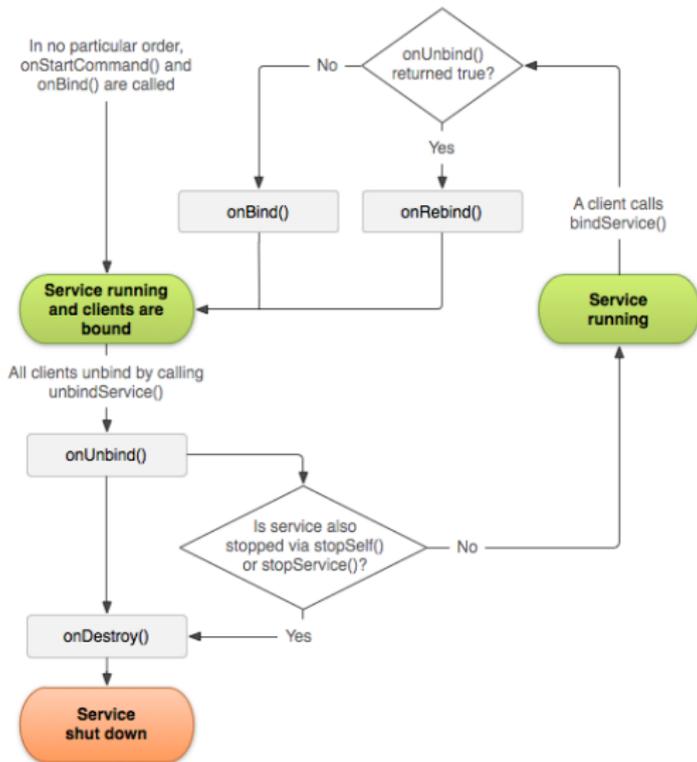
- There are three possible ways to implement a bound service:
    - By extending the `Binder` class. It works only when the clients are local and run in the same process though.
    - By using a `Messenger`, that will provide the interface for your service to remote processes. However, it does not perform 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

In no particular order, onStartCommand() and onBind() are called

No — onUnbind() returned true?

Yes

onBind()

onRebind()

A client calls bindService()

Service running and clients are bound

Service running

All clients unbind by calling unbindService()

onUnbind()

Is service also stopped via stopSelf() or stopService()?

No

onDestroy()

Yes

Service shut down

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

```java
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

```java
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.

- ▶ 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

```xml
<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

Credits: http://developer.android.com
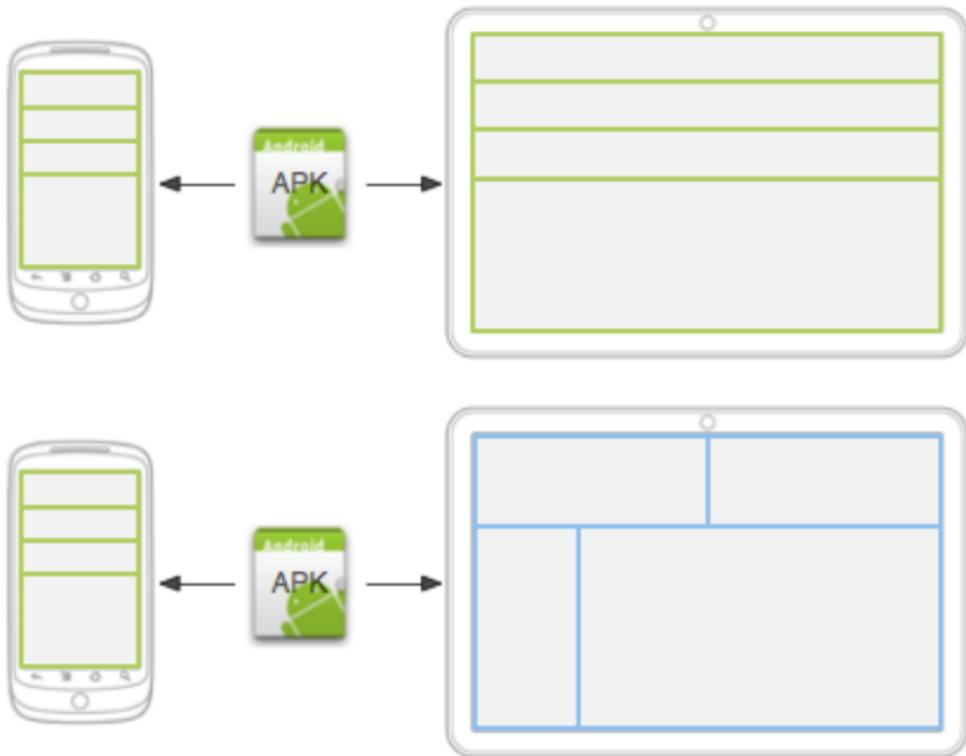
Alternative Resources

▶ 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`
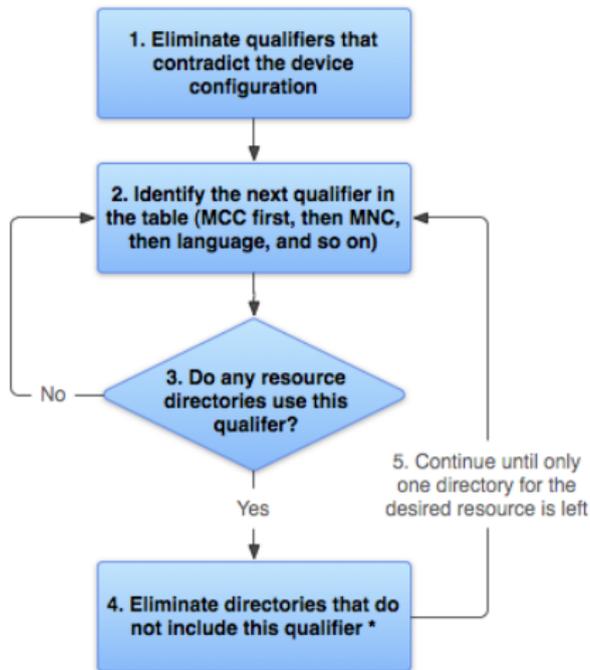
Credits: http://developer.android.com

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
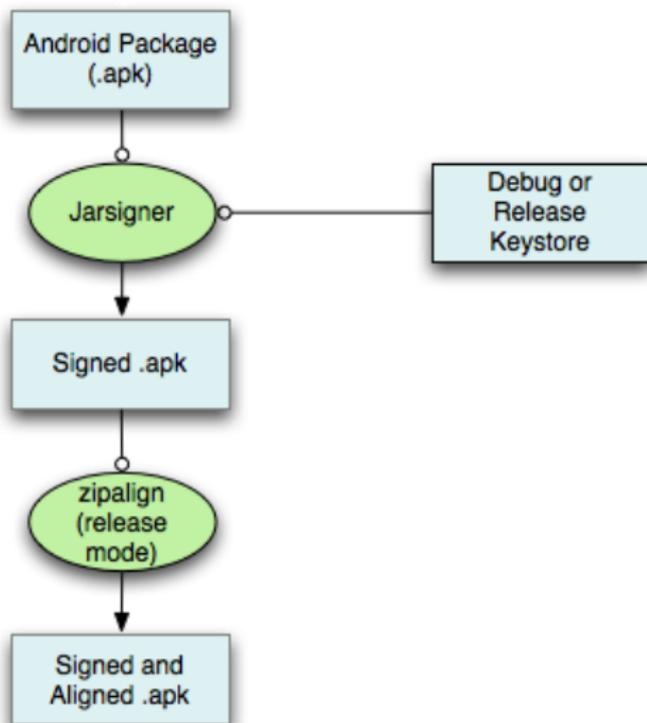- `resources.arsc` is the resources table. It keeps track of the package resources, associated IDs and packages
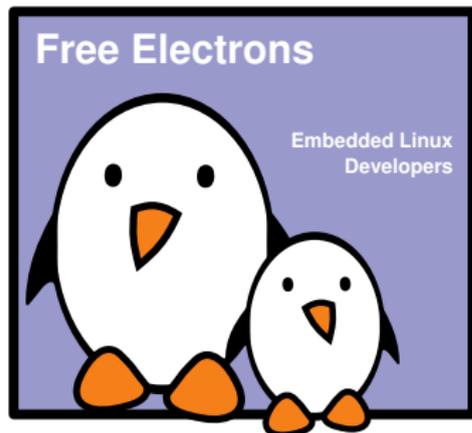
Credits: http://developer.android.com

Credits: http://developer.android.com
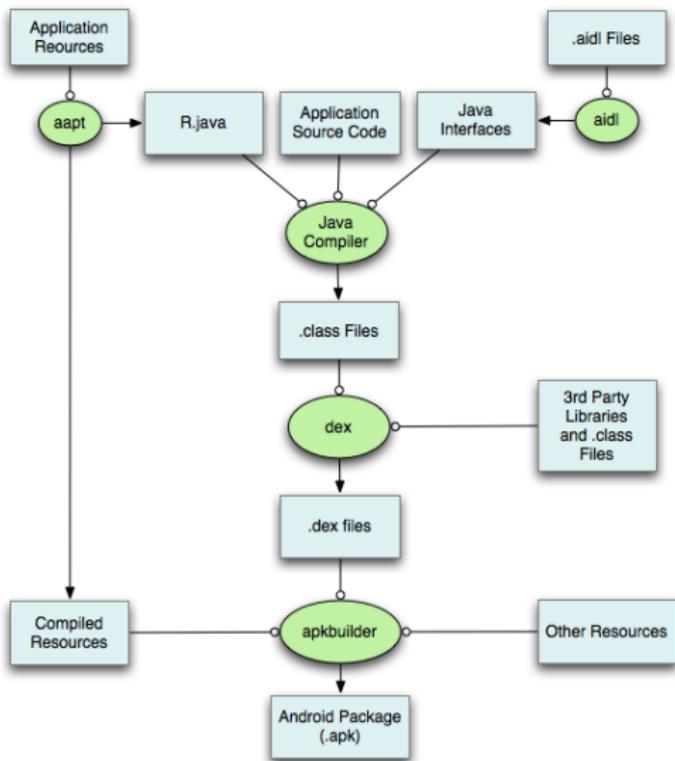
Advices and Resources

Maxime Ripard
Free Electrons

Free Electrons

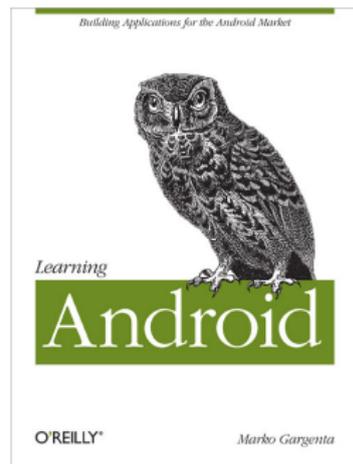Embedded Linux
Developers

Embedded Android, July 2012 (Expected)

- By Karim Yaghmour, O'Reilly
- A good reference book and guide on all hidden and undocumented Android internals
- Our rating: 3 stars
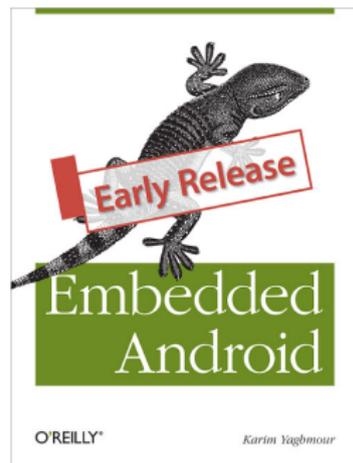
Learning Android, March 2011

- ▶ 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/

Conferences

Useful conferences featuring Android topics:

- Android Builders Summit:
  https://events.linuxfoundation.org/events/
  android-builders-summit
  Organized by the Linux Foundation in California (in the Silicon Valley) in early Spring. Many talks about the whole Android stack. Presentation slides are freely available on the Linux Foundation website.

- Embedded Linux Conference:
  http://embeddedlinuxconference.com/
  Organized by the Linux Foundation: California (Silicon Valley, Spring), in Europe (Fall). Mostly about kernel and userspace Linux development in general, but always some talks about Android. Presentation slides freely available

- Don't miss our free conference videos on http://free-electrons.com/community/videos/conferences/!