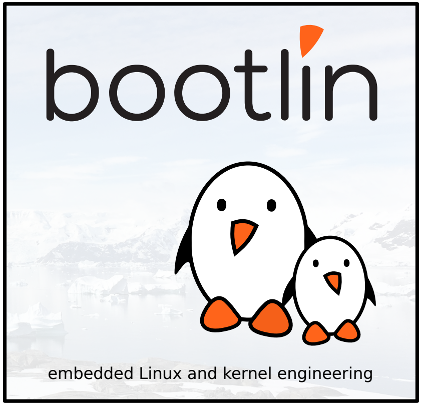
Android System Development

Maxime Ripard
maxime.ripard@bootlin.com



Maxime Ripard

- Embedded Linux engineer and trainer at Bootlin since 2011
  - Embedded Linux development: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux training, Linux driver development training and Android system development training, with materials freely available under a Creative Commons license.
  - `https://bootlin.com`
- Contributor to various open-source projects: Barebox, Linux, Buildroot
- Living in Toulouse, south west of France

Bootlin

- Bootlin, specialized in Embedded Linux, since 2005
- Strong emphasis on community relation
- **Training**
  - *Embedded Linux system development*
  - *Linux kernel and device driver development*
  - Upcoming public sessions in Avignon, Lyon and Toulouse, or sessions at customer location
  - All training materials freely available under a Creative Commons license.
- **Development and consulting**
  - Board Support Package development or improvement
  - Kernel and driver development
  - Embedded Linux system integration
  - Power-management, boot-time, performance audits and improvement
  - Embedded Linux application development

Bootlin customers

ALSTOM Transport

ATMEL Everywhere You Are®

freescale™ semiconductor

MARVELL®

Adeneo Embedded

IKUSI

Nokia Siemens Networks

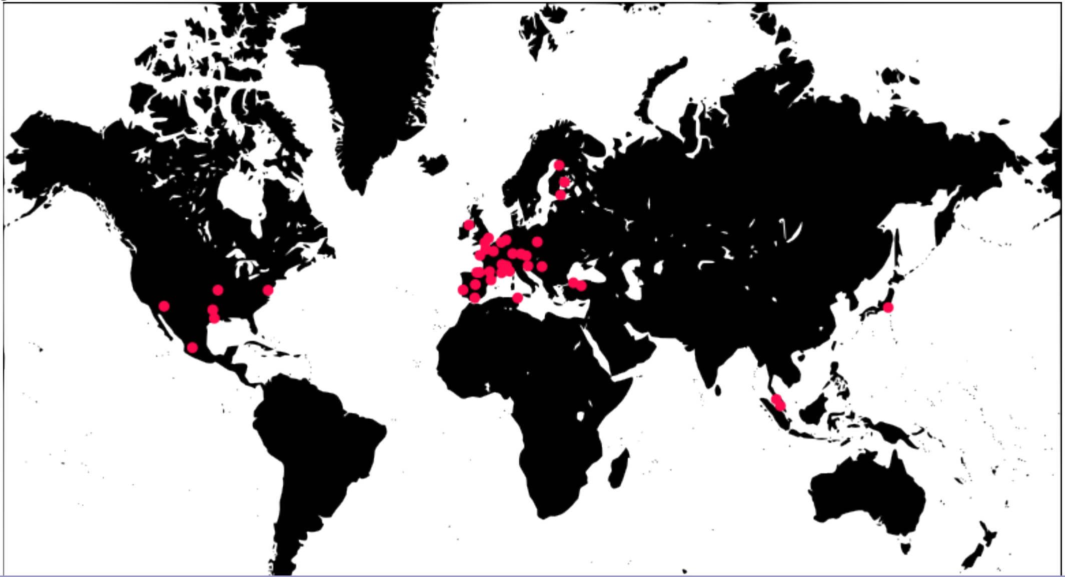
Alcatel·Lucent

SIEMENS

Continental

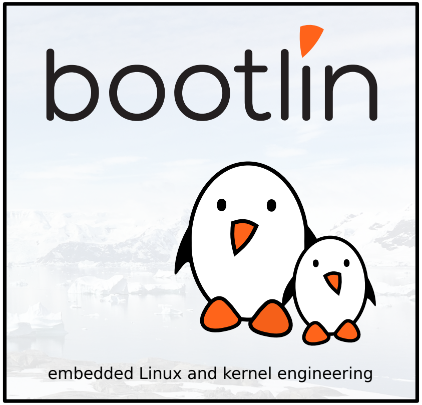
orange™

Bootlin training



Introduction to Android

Maxime Ripard
maxime.ripard@bootlin.com



Features

Features

- All you can expect from a modern mobile OS:
  - Application ecosystem, allowing to easily add and remove applications and publish new features across the entire system
  - Support for all the web technologies, with a browser built on top of the well-established 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
- Andy Rubin, founder of Android, Inc was also CEO of Danger, Inc, a company producing one of the early smartphones, the Sidekick

Opening Up

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

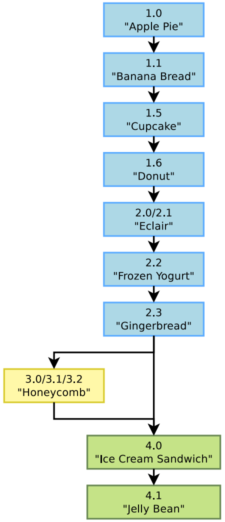
Android Open Source Project (AOSP)

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

Android Releases

- Each new version is given a dessert name
- Released in alphabetical order
- Last releases:
  - Android 3.X Honeycomb
  - Android 4.0 Ice Cream Sandwich
  - Android 4.1/4.2 Jelly Bean

Android Versions



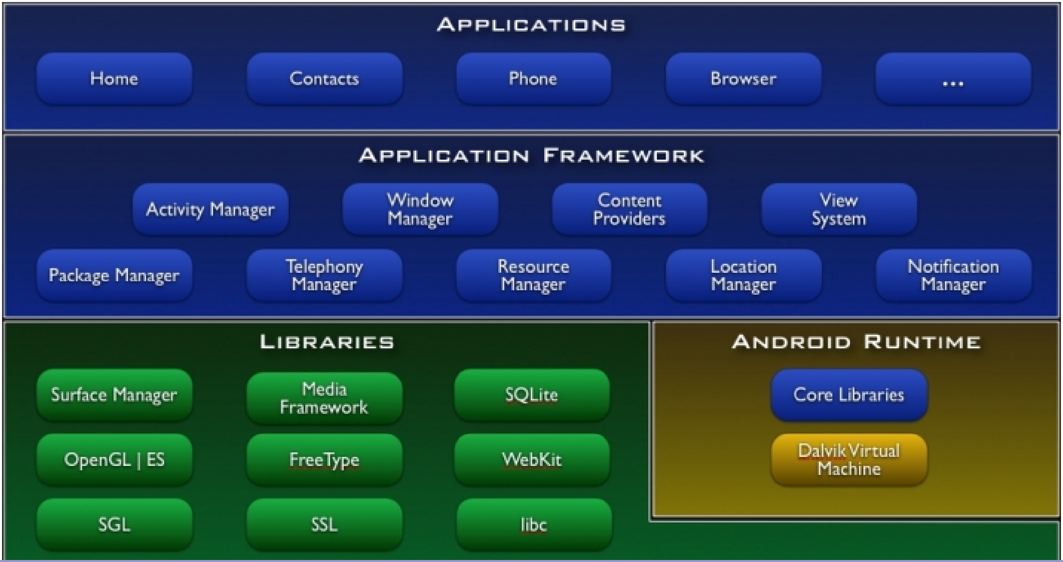
🟦 Phones only

🟨 Tablets only

🟩 Tablets and phones

Architecture

Architecture



The Linux Kernel

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

Android Libraries

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

Android Framework

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

Android Applications

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

Hardware Requirements for Linux

Processor architecture

- The Linux kernel and most other architecture-dependent component **support a wide range of 32 and 64 bits architectures**
  - x86 and x86_64, as found on PC platforms, but also embedded systems (multimedia, industrial)
  - ARM, with hundreds of different SoC (multimedia, industrial)
  - PowerPC (mainly real-time, industrial applications)
  - MIPS (mainly networking applications)
  - SuperH (mainly set top box and multimedia applications)
  - Blackfin (DSP architecture)
  - Microblaze (soft-core for Xilinx FPGA)
  - Coldfire, SCore, Tile, Xtensa, Cris, FRV, AVR32, M32R
- Both **MMU and no-MMU architectures are supported**, even though no-MMU architectures have a few limitations.
- Linux is **not designed for small microcontrollers**.
- Besides the toolchain, the bootloader and the kernel, all other components are **generally architecture-independent**

RAM and storage

- **RAM**: a very basic Linux system can work within 8 MB of RAM, but a more realistic system will usually require at least 32 MB of RAM. Depends on the type and size of applications.
- **Storage**: a very basic Linux system can work within 4 MB of storage, but usually more is needed.
  - *flash storage* is supported, both NAND and NOR flash, with specific filesystems
  - *Block storage* including SD/MMC cards and eMMC is supported
- Not necessarily interesting to be too restrictive on the amount of RAM/storage: having flexibility at this level allows to re-use as many existing components as possible.

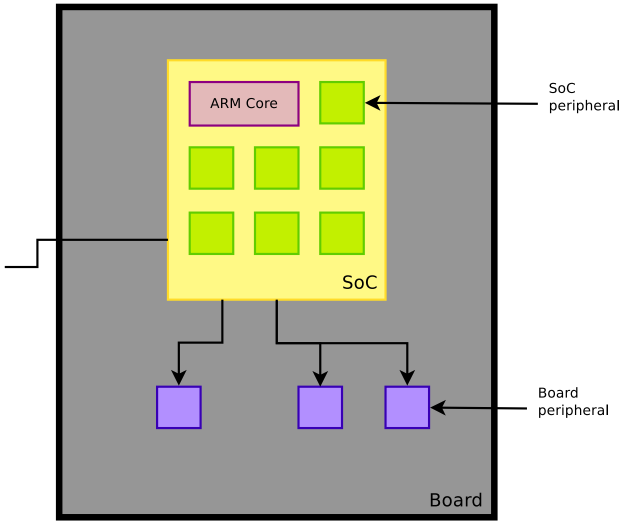
Communication

- The Linux kernel has support for many common communication buses
  - I2C
  - SPI
  - CAN
  - 1-wire
  - SDIO
  - USB
- And also extensive networking support
  - Ethernet, WiFi, Bluetooth, CAN, etc.
  - IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
  - Firewalling, advanced routing, multicast

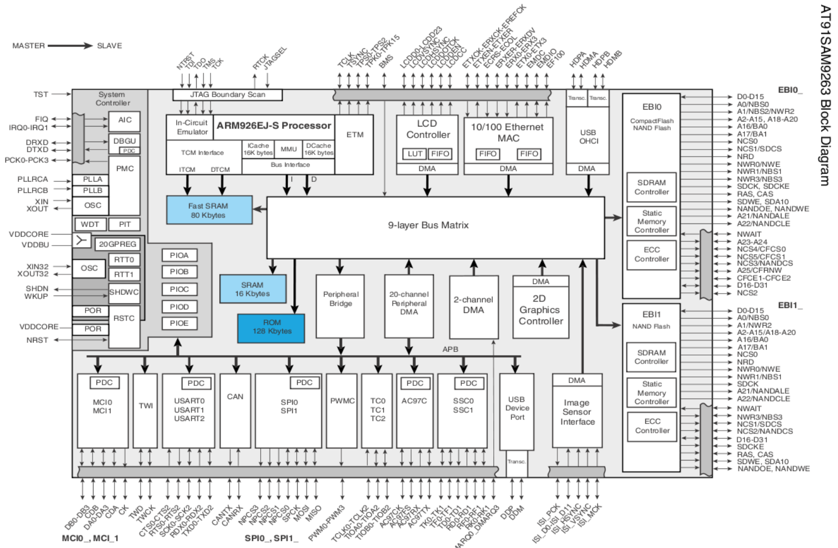
ARM and System-on-Chip

- ARM is one of the most popular architectures used in embedded Linux systems
- ARM designs **CPU cores** (instruction sets, caches, MMU, etc.) and sells the design to licensees
- The licensees are **founders** (Texas Instruments, Freescale, ST Ericsson, Atmel, etc.), they integrate an ARM core with many peripherals, into a chip called a **SoC**, for System-on-chip
- Each founder provides different models of SoC, with **different combination of peripherals**, power, power consumption, etc.
- The concept of SoC allows to reduce the number of peripherals needed on the board, and therefore the cost of designing and building the board.
- **Linux supports SoCs from most vendors, but not all**, and not with the same level of functionality.

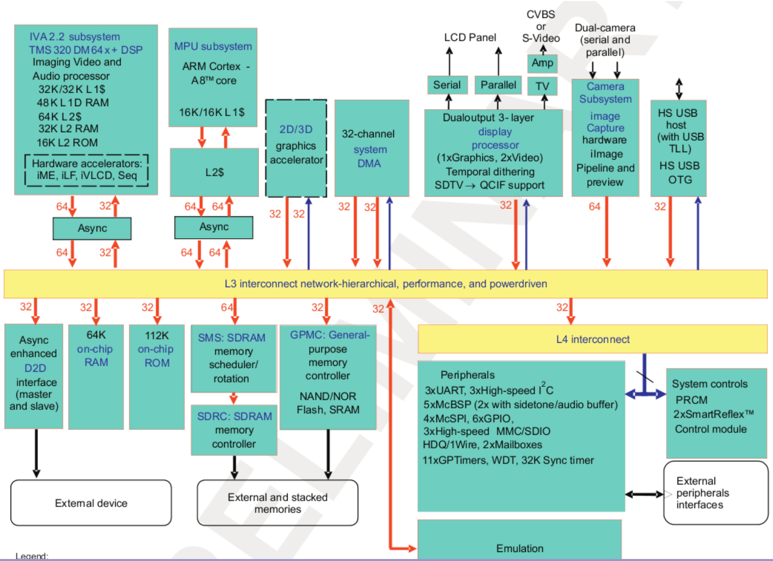
ARM and System-on-Chip



Atmel AT91SAM9263



Texas Instruments OMAP3430


IVA2.2 subsystem
TMS 320 DM64x+ DSP
Imaging Video and Audio processor
32K/32K L1$
48K L1D RAM
64K L2$
32K L2 RAM
16K L2 ROM
Hardware accelerators: iME, iLF, iVLCD, Seq
MPU subsystem
ARM Cortex - A8™ core
16K/16K L1$
L2$
Async
Async
2D/3D graphics accelerator
32-channel system DMA
LCD Panel
CVBS or S-Video
Dual-camera (serial and parallel)
Serial
Parallel
Amp
TV
Dualoutput 3-layer display processor (1xGraphics, 2xVideo) Temporal dithering SDTV → QCIF support
Camera Subsystem
image Capture hardware
iImage Pipeline and preview
HS USB host (with USB TLL)
HS USB OTG
L3 interconnect network-hierarchical, performance, and powerdriven
Async enhanced D2D interface (master and slave)
64K on-chip RAM
112K on-chip ROM
SMS: SDRAM memory scheduler/ rotation
SDRC: SDRAM memory controller
GPMC: General-purpose memory controller
NAND/NOR Flash, SRAM
L4 interconnect
Peripherals
3xUART, 3xHigh-speed I²C
5xMcBSP (2x with sidetone/audio buffer)
4xMcSPI, 6xGPIO,
3xHigh-speed MMC/SDIO
HDQ/1Wire, 2xMailboxes
11xGPTimers, WDT, 32K Sync timer
System controls
PRCM
2xSmartReflex™
Control module
External device
External and stacked memories
External peripherals interfaces
Emulation
Legend:
64
32
32
32
64
64
64
64
32
32
32
32
32
32
64
32
32
32
32
64
32
32
32

Type of hardware platforms

- **Evaluation platforms** from the SoC vendor. Usually expensive, but many peripherals are built-in. Generally unsuitable for real products.
- **Component on Module**, a small board with only CPU/RAM/flash and a few other core components, with connectors to access all other peripherals. Can be used to build end products for small to medium quantities.
- **Community development platforms**, a new trend to make a particular SoC popular and easily available. Those are ready-to-use and low cost, but usually have less peripherals than evaluation platforms. To some extent, can also be used for real products.
- **Custom platform**. Schematics for evaluation boards or development platforms are more and more commonly freely available, making it easier to develop custom platforms.

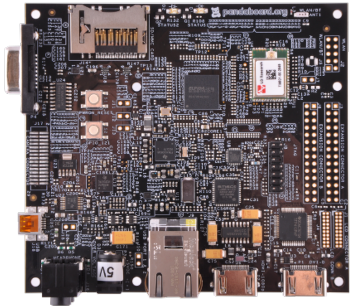
Gumstix

- **TI OMAP3530 (600 MHz, ARM Cortex-A8, PowerVR SGX, DSP)**
- Component on Module
- 256 MB RAM
- 256 MB NAND (optional)
- Bluetooth, WiFi (optional)
- uSD
- $115-229
- Development boards available at $ 49-229, with many peripherals: LCD, Ethernet, UART, SPI, I2C, etc.
- `http://www.gumstix.com`

Overero FE
Ships with:
2 x u.fl antennas for Bluetooth and 802.11g
4 x retaining spacers
Power supply not included

Pandaboard ES

- **TI OMAP 4460 (Dual-Core 1.2 GHz, cortex A9, PowerVR GPU, DSP)**
- Community development platform
- 1 GiB RAM
- SD slot, Ethernet, WiFi, Bluetooth, USB OTG
- HDMI, S-Video, Camera, audio
- $ 149
- `http://pandaboard.org`

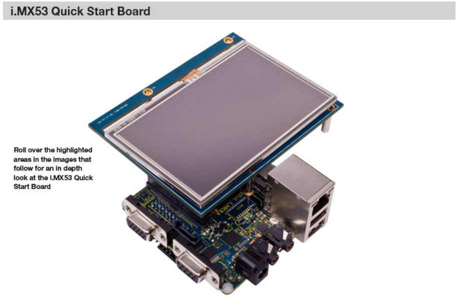


Snowball

- **ST Ericsson AP9500 (dual cortex A9, MALI GPU)**
- Community development platform
- 1 GB RAM
- 4-8 GB eMMC, microSD
- HDMI, S-Video, audio
- WiFi, Bluetooth, Ethernet
- Accelerometer, Magnetometer, Gyrometer, GPS
- Expansion: USB, I2C, SPI, LCD, UART, GPIO, etc.
- 169 € to 244 €
- `http://igloocommunity.org`

Freescale Quick Start

- **Freescale I.MX53 (1 GHz Cortex A8)**
- Community development platform
- 1 GB RAM
- 4-8 GB eMMC, microSD
- LVDS, LCD, VGA, HDMI, audio
- Accelerometer, SD/MMC, microSD, SATA, Ethernet, USB
- Expansion: I2C, SPI, SSI, LCD, Camera
- 149 €


i.MX53 Quick Start Board

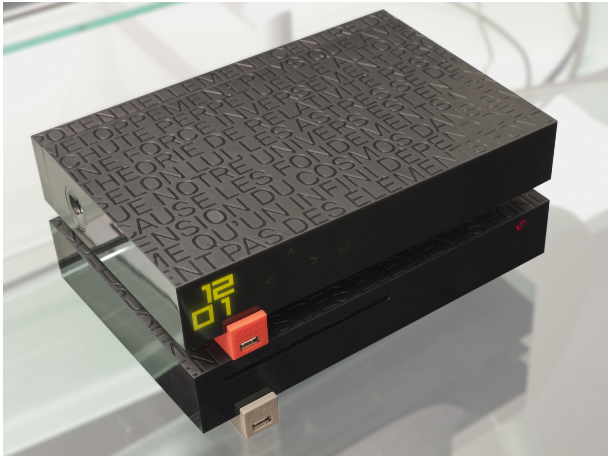
Roll over the highlighted areas in the images that follow for an in depth look at the i.MX53 Quick Start Board

Criteria for choosing the hardware

- Make sure the hardware you plan to use is already supported by the Linux kernel, and has an open-source bootloader, especially the SoC you're targeting.
- Having support in the official versions of the projects (kernel, bootloader) is **a lot better**: quality is better, and new versions are available.
- Some SoC vendors and/or board vendors do not contribute their changes back to the Linux kernel. Ask them to do so, or use another product if you can. A good measurement is to see the delta between their kernel and the official one.
- Between properly supported hardware in the official Linux kernel and poorly-supported hardware, there will be huge differences in development time and cost.

Embedded Systems using Linux

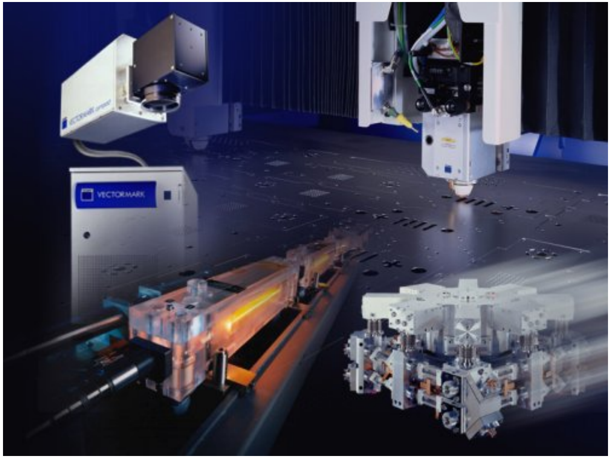
Consumer device: Internet box



Consumer device: Television

Professional device: point of sale terminal

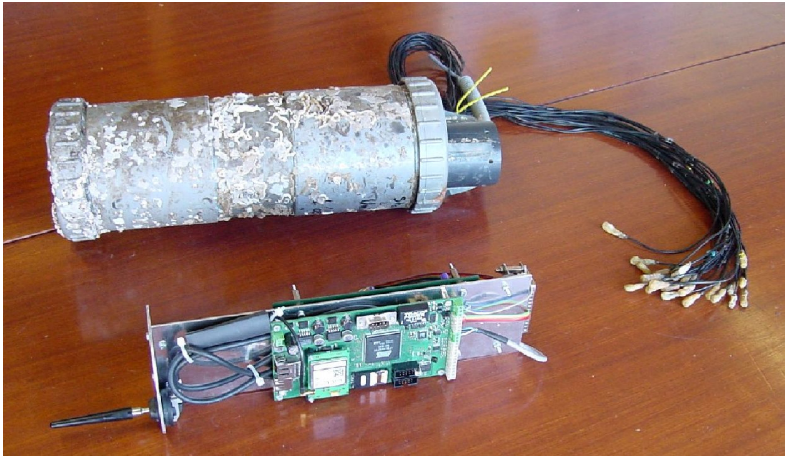
Industrial system: laser cutter



Industrial system: wind turbine

Industrial system: cow milking

Industrial system: sea pollution detection system



Industrial system: viticulture machine

Critical Systems: Nuclear Reactor

Critical Systems: Rockets

Hardware Requirements for Android

SoC requirements

- Since Android in itself is quite huge, the hardware required is quite different.
- First, the only architecture officially supported by Google is ARMv7 (basically, all the SoCs based on the Cortex-A). The CPU has to be powerful enough as well (the typical setup on recent releases is a dual-core CPUs running at more than 1GHz
  - Other architectures like x86 and MIPS are supported by third-party ports
- You also need to have a powerful enough GPU with OpenGL ES support. Latest versions of Android require the 3D hardware acceleration

Storage and RAM needed

- The required RAM size is also quite huge, 340MB are required for the kernel and userspace memory
- Required storage is quite huge as well. An image of the system is around 200-300MB, and you must have 350MB of data space for the user plus 1GB of shared storage for the applications. Google recommends to use block devices for storage and not flash devices.
- The shared space has to be accessible from a host computer by some way, like NFS, USB Mass Storage, MTP, etc.

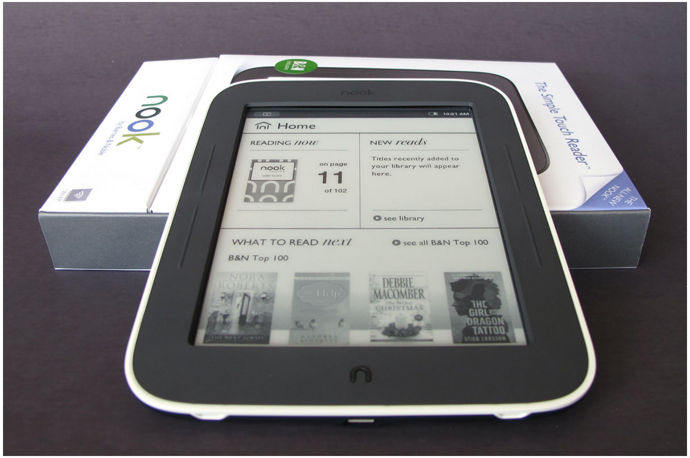
External Peripherals 1/2

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

External Peripherals 2/2

- Sensors are not mandatory, but depending of the class of sensors, they are:
  - Recommended
    - Accelerometer
    - Magnetometer
    - GPS
    - Gyroscope
  - Optional
    - Barometer
    - Photometer
    - Proximity Sensor
  - Deprecated
    - Thermometer (and only to measure CPU temperature)

Unusual Android Devices: Nook E-Book Reader



Unusual Android Devices: Watch

Unusual Android Devices: Portable Console

Unusual Android Devices: Microwave Oven

Unusual Android Devices: Treadmill

When to choose Android

- All of the requirements listed above are only if you want to be eligible to the Android Play Store
- If you don't want to get the store, you can obviously ignore these
- However, Android really makes sense in a system that has at least:
  - A large screen
  - A powerful SoC, with plenty of RAM, storage space (around 512MB) and a decent GPU
- This is not an advisable choice when you want to build a headless system, or a cheap system with limited resources
- In this case, a regular Linux system is definitely more appropriate. It will save you engineering costs, reduce the price of your hardware, and bring the same set of features you could expect from a headless Android

Android Source Code and Compilation

Maxime Ripard
maxime.ripard@bootlin.com

bootlin
embedded Linux and kernel engineering

How to get the source code

Source Code Location

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

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.

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

Android Source Code and Compilation

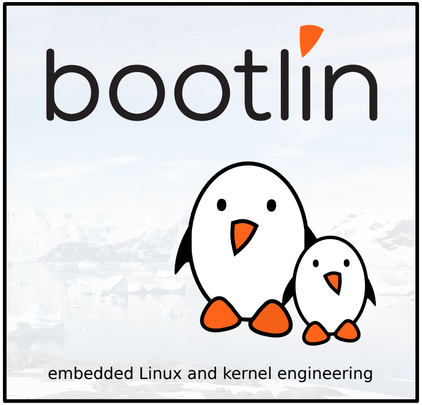
Contribute

Gerrit

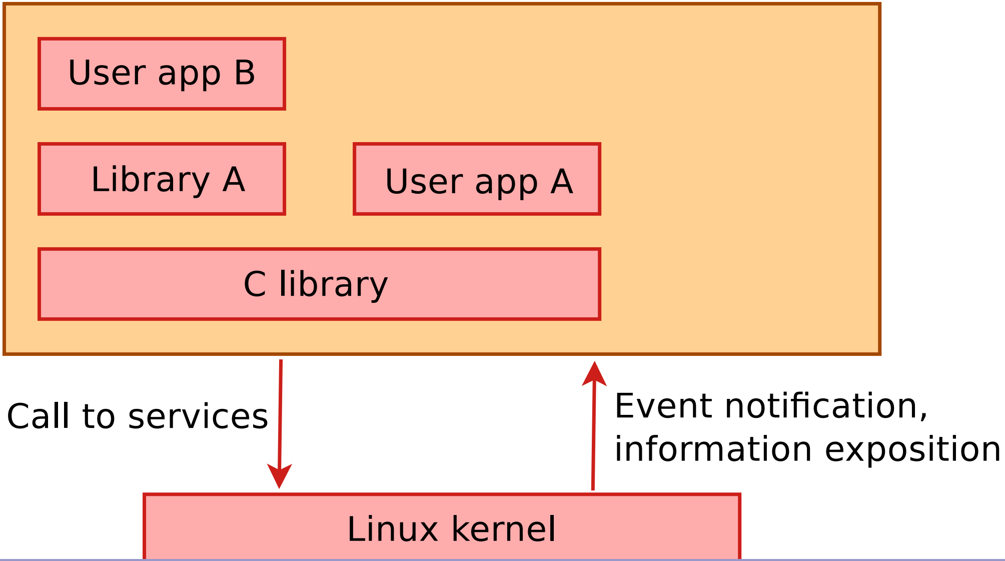
- Google also developed for the Android development process a tool to manage projects and ease code reviews.
- It once again uses Git to do so and Repo is also built around it so that you can easily contribute to Android
- To do so, start a new branch with `repo start <branchname>`
- Do your usual commits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

Linux kernel introduction

Maxime Ripard
maxime.ripard@bootlin.com



Linux kernel in the system



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

- See the `arch/` directory in the kernel sources
- Minimum: 32 bit processors, with or without MMU, and `gcc` support
- 32 bit architectures (`arch/` subdirectories)
  Examples:
  `arm, avr32, blackfin, m68k, microblaze, mips, score, sparc, um`
- 64 bit architectures:
  Examples: `alpha, arm64, ia64, sparc64, tile`
- 32/64 bit architectures
  Examples: `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`, usually mounted on `/proc`:
    Operating system related information (processes, memory management parameters...)
  - `sysfs`, usually mounted on `/sys`:
    Representation of the system as a set of devices and buses. Information about these devices.

Changes introduced in the Android Kernel

Maxime Ripard
maxime.ripard@bootlin.com

Wakelocks

Power management basics

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

Wakelocks

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

Wakelocks API

- Kernel Space API

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

- User-Space API

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

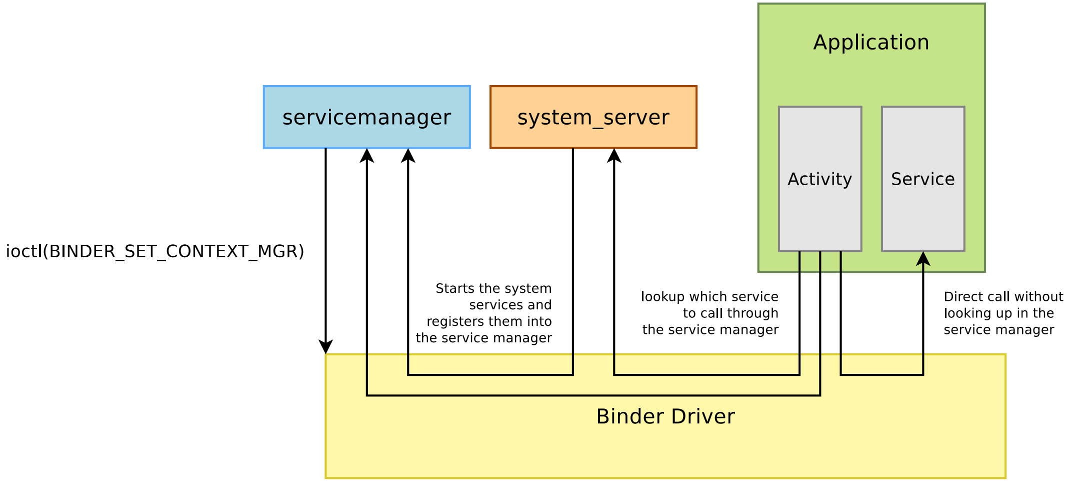
The Android Kernel

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.

Binder


Application
servicemanager
system_server
Activity
Service
ioctl(BINDER_SET_CONTEXT_MGR)
Starts the system services and registers them into the service manager
lookup which service to call through the service manager
Direct call without looking up in the service manager
Binder Driver

The Android Kernel

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

The Android Kernel

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 behavior 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

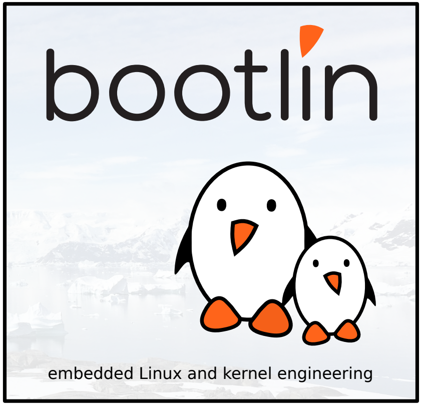
Various Drivers and Fixes

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 Filesystem

Maxime Ripard
maxime.ripard@bootlin.com



Android Filesystem

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.

Root filesystem

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

```
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown block(0,0)
```

Location of the root filesystem

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

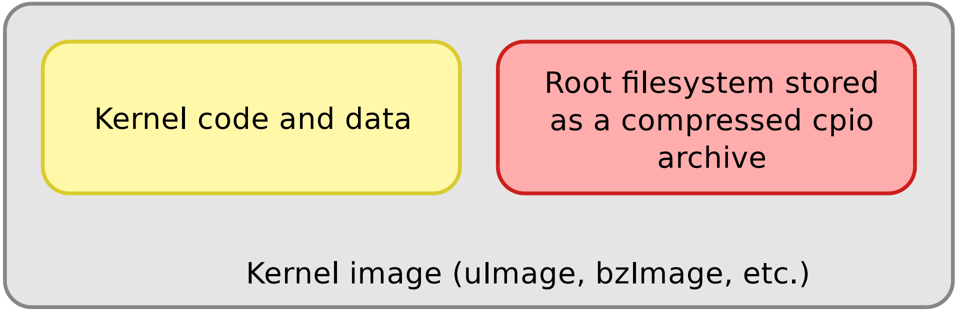
Mounting rootfs from storage devices

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

rootfs in memory: initramfs (1)

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

rootfs in memory: initramfs (2)


Kernel code and data
Root filesystem stored as a compressed cpio archive
Kernel image (uImage, bzImage, etc.)

rootfs in memory: initramfs (3)

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

Android Filesystem

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 common 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

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
  ```
- `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 `/latest/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
  ```

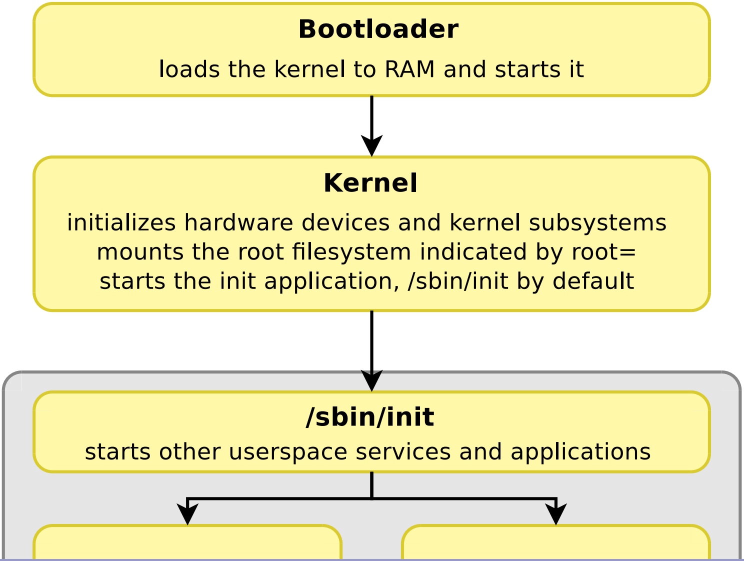
Android Filesystem

Minimal filesystem

Basic applications

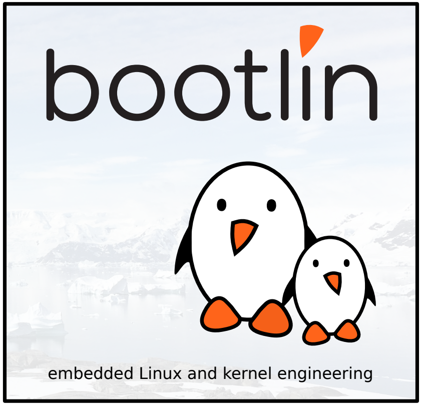
- In order to work, a Linux system needs at least a few applications
- An `init` application, which is the first userspace application started by the kernel after mounting the root filesystem
  - The kernel tries to run `/sbin/init`, `/bin/init`, `/etc/init` and `/bin/sh`.
  - If none of them are found, 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

Overall booting process


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

Android Build System

Maxime Ripard
maxime.ripard@bootlin.com



Android Build System

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
- `make clean` only deletes that `out` directory

Add a New Module

Modules

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

Hello World

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

LOCAL_SRC_FILES = hello_world.c
LOCAL_MODULE = HelloWorld

LOCAL_MODULE_TAGS = optional

include $(BUILD_EXECUTABLE)
```

Hello World

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

Tags

- Tags are used to define several sets of modules to be built through the build variant selected by `lunch`
- We have 3 build variants:
  - `user`
    - Installs modules tagged with `user`
    - Installs non-packaged modules that have no tags specified
    - ADB is disabled by default
  - `userdebug` is `user` plus
    - Installs modules tagged with `debug`
    - ADB is enabled by default
  - `eng` is `userdebug`, plus
    - Installs modules tagged as `eng` and `development`
    - `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`

Make and clean a module

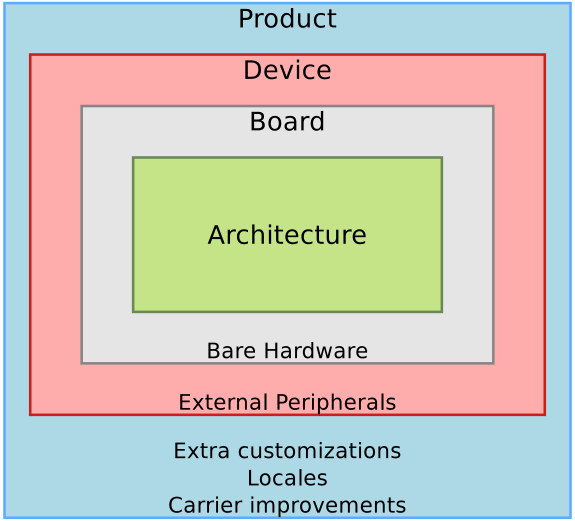
- To build a module from the top directory, just do `make ModuleName`
- However, building a simple module won't regenerate a new image. This is just useful to make sure that the module builds. You will have to do a full `make` to have an image that contains your module
- Actually, a full `make` will 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.
- You can also get the list of the modules to be built with the `make modules` target

Add a New Product

Defining new products

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

Product, devices and boards



Minimal Product Declaration

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

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

Copy files to the target

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

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

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

Add a package to the build for this product

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

PRODUCT_PACKAGES += FooBar

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

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

Overlays

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

Add a package to the build for this product

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

PRODUCT_PACKAGES += FooBar

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

DEVICE_PACKAGE_OVERLAYS := device/mybrand/mydevice/overlay

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

Board Definition

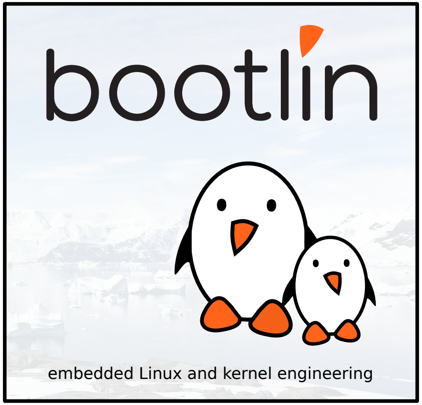
- You will also need a `BoardConfig.mk` file along with the product definition
- While the product definition was mostly about the build system in itself, the board definition is more about the hardware
- You can have a full working example in `device/samsung/crespo/BoardConfigCommon.mk`
- However, this 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

Minimal Board Definition

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

USE_CAMERA_STUB := true
```

Android Native Layer

Maxime Ripard
maxime.ripard@bootlin.com



Bionic

Bionic

- 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.
- 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

Android Native Layer

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`

Android Native Layer

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 hotplugging, with basic permissions rules for device files, and actions at device plugging and unplugging
  - it monitors the services it started, so that if they crash, it can restart them
  - it monitors system properties so that you can take actions when a particular one is modified

Init part

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

init.rc file interpretation

- Uses a unique syntax, based on events
- There usually are several init configuration files, `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
- This name is either obtained by reading the file `/proc/cpuinfo` or from the `androidboot.hardware` kernel parameter
- Most of the customizations should therefore go to the platform-specific configuration file rather than to the generic one

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`

Properties

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

Modifying Properties

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

Permissions on the Properties

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

```
/* White list of permissions for setting property services. */
struct {
    const char *prefix;
    unsigned int uid;
    unsigned int gid;
} property_perms[] = {
    { "net.rmnet0.",      AID_RADIO,    0 },
    { "net.dns",          AID_RADIO,    0 },
    { "net.",             AID_SYSTEM,   0 },
    { "dhcp.",            AID_SYSTEM,   0 },
    { "log.",             AID_SHELL,    0 },
    { "service.adb.root", AID_SHELL,    0 },
    { "persist.security.", AID_SYSTEM,   0 },
    { NULL, 0, 0 }
};
```

Special Properties

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

Android Native Layer

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

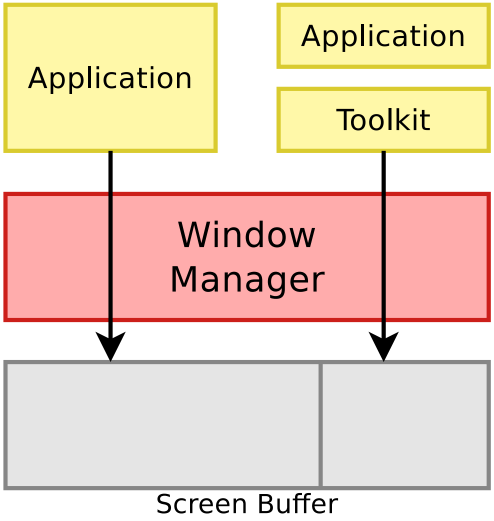
- `rild` is the Radio Interface Layer Daemon
- This daemon drives the telephony stack, both voice and data communication
- When using the voice mode, talks directly to the baseband, but when issuing data transfers, relies on the kernel network stack
- It can handle two types of commands:
  - *Solicited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

Others

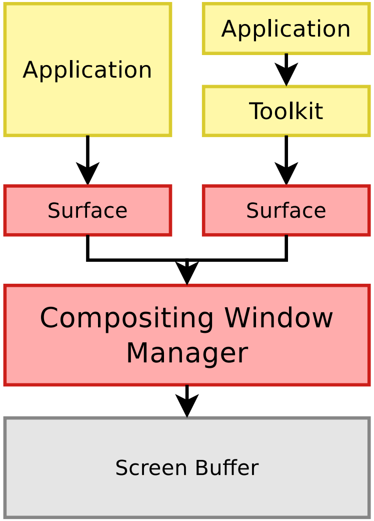
- netd
  - `netd` manages the various network connections: Bluetooth, Wifi, USB
  - Also takes any associated actions: detect new connections, set up the tethering, etc.
  - It really is an equivalent to NetworkManager
  - On a security perspective, it also allows to isolate network-related privileges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

SurfaceFlinger and PixelFlinger

Introduction to graphical stacks



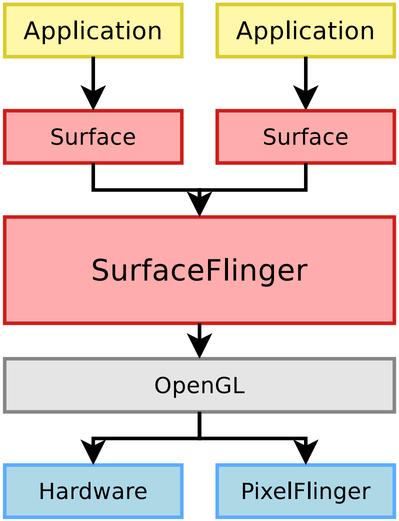
Compositing window managers



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.

SurfaceFlinger and PixelFlinger

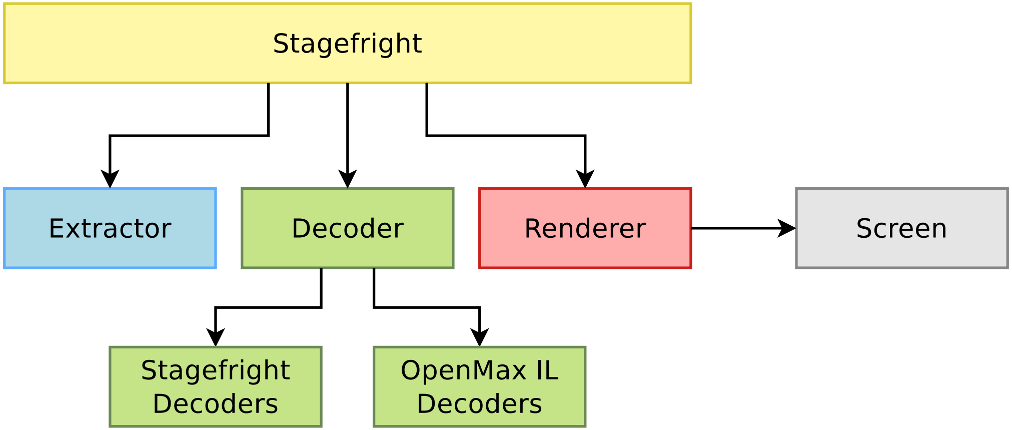


Stagefright

Stagefright

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

StageFright Architecture



StageFright plugins

- To add support for a new format, you need to:
  - Develop a new Extractor class, if the container is not supported yet.
  - Develop a new Decoder class, that implements the interface needed by the StageFright core to read the data.
  - Associate the mime-type of the files to read to your new Decoder in the `OMXCodec.cpp` file, in the `kDecoderInfo` array.
    - $\rightarrow$ 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" },
};
```

Android Native Layer

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

Android Native Layer

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`

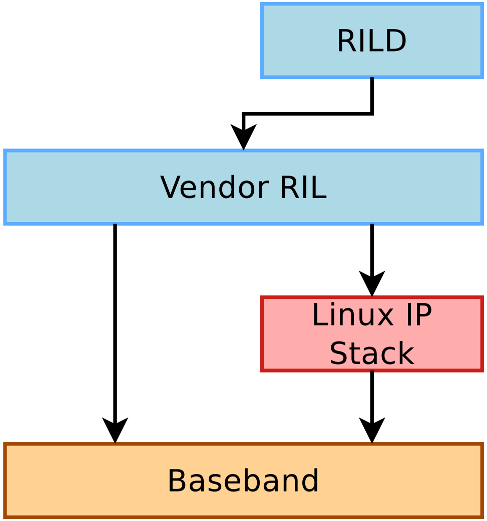
Various layers

- 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 communication between the baseband and `rild`
  - You can set the name of the library with the `rild.lib` and `rild.libargs` properties to find the library
  - `hardware/ril/include/telephony/ril.h`

Example: rild



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

C Code

```c
#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++

Java Code

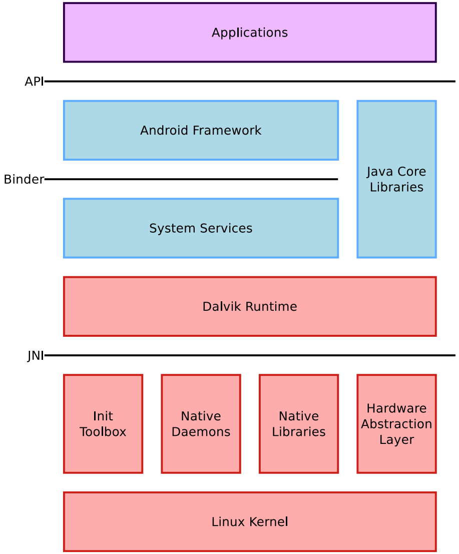
```
package com.example;

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

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

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

Android Framework and Applications

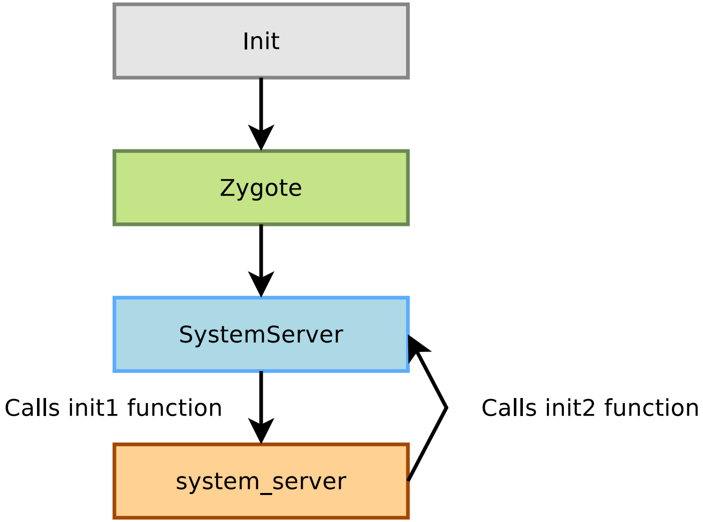
Maxime Ripard
maxime.ripard@bootlin.com

Service Manager and Various Services

Whole Android Stack



System Server boot



The first step: `system_server.c`

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

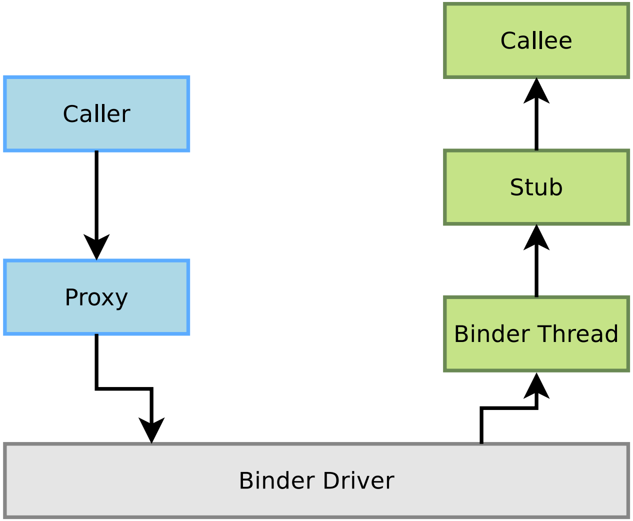
IPCs

- 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, several IPC mechanisms are used:
  - Signals
  - Semaphores
  - Sockets
  - Message queues
  - Pipes
  - Shared memory
- Android, however, uses mostly:
  - Binder
  - Ashmem and Sockets

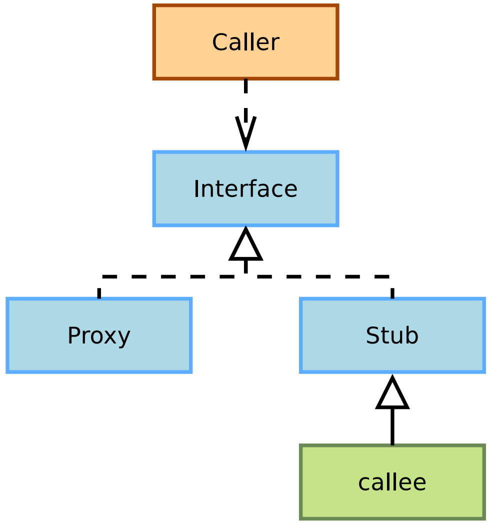
Binder

- 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, that is used to distinct components from a given process, or to enforce security

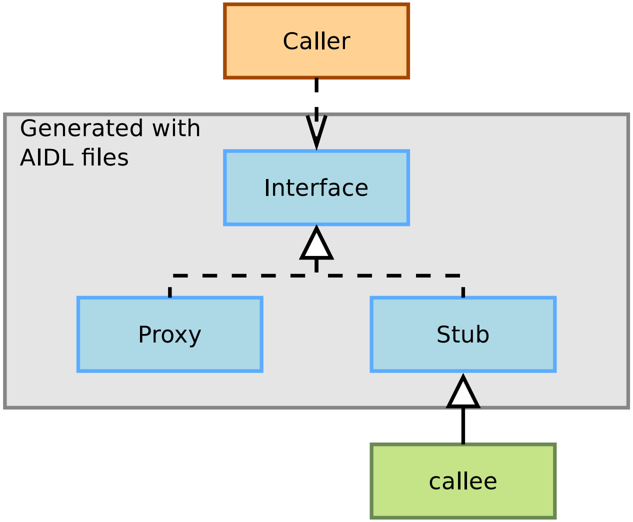
Binder Mechanism



Binder Implementation 1/2



Binder Implementation 2/2



Android Interface Definition Language (AIDL)

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

AIDLs HelloWorld

```
package com.example.android;

interface IRemoteService {
    void HelloPrint(String aString);
}
```

Intents

- Intents are a high-level use of Binder
- They describe the intention to do something
- They are used extensively across Android
  - Activities, Services and BroadcastReceivers are started using intents
- Two types of intents:

explicit	The developer designates the target by its name
implicit	There is no explicit target for the Intent. The system will find the best target for the Intent by itself, possibly asking the user what to do if there are several matches

Various Java Services

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 life cycle
  - 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

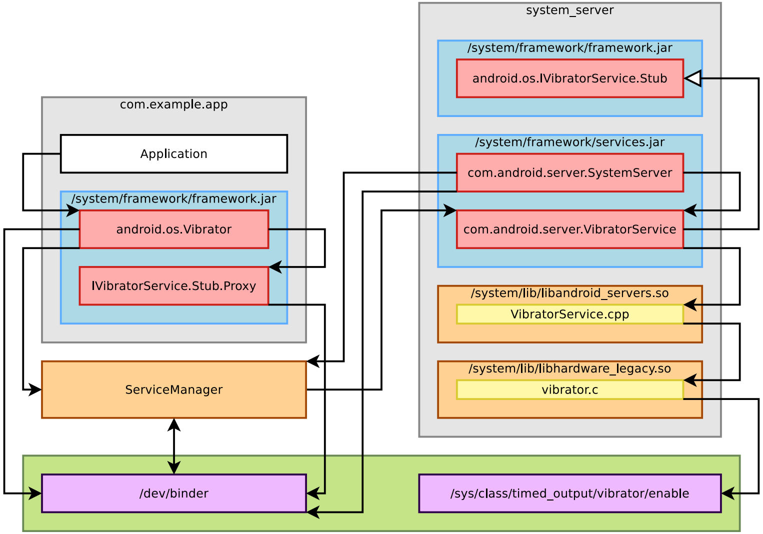
AlarmManager

- 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

Example: Vibrator Service


system_server
/system/framework/framework.jar
android.os.IVibratorService.Stub
/system/framework/services.jar
com.android.server.SystemServer
com.android.server.VibratorService
/system/lib/libandroid_servers.so
VibratorService.cpp
/system/lib/libhardware_legacy.so
vibrator.c
com.example.app
Application
/system/framework/framework.jar
android.os.Vibrator
IVibratorService.Stub.Proxy
ServiceManager
/dev/binder
/sys/class/timed_output/vibrator/enable

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

Developing and Debugging with ADB

Maxime Ripard
maxime.ripard@bootlin.com

bootlin

embedded Linux and kernel engineering

Android Debug Bridge

Introduction

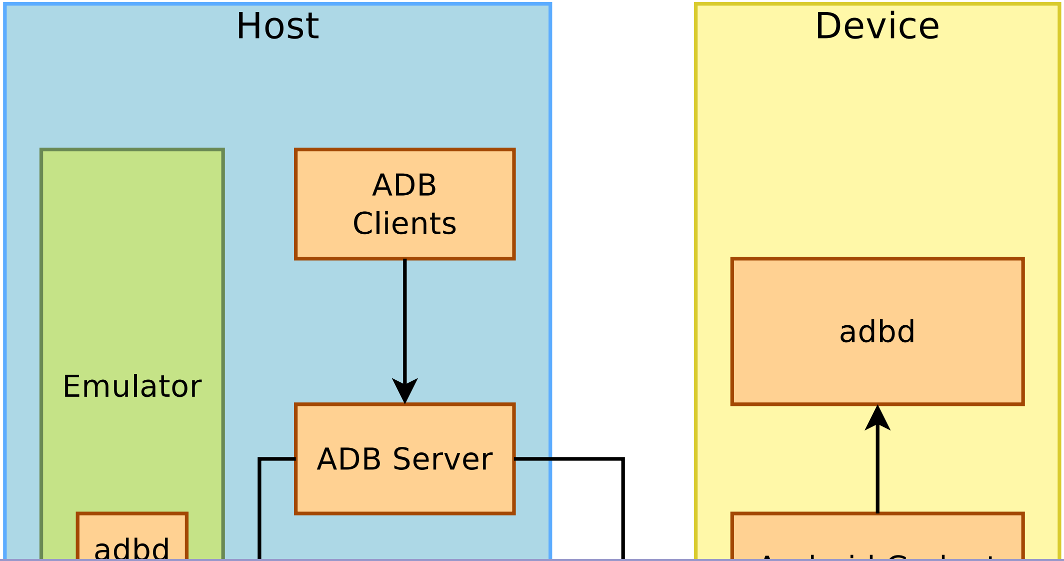
ADB

- 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



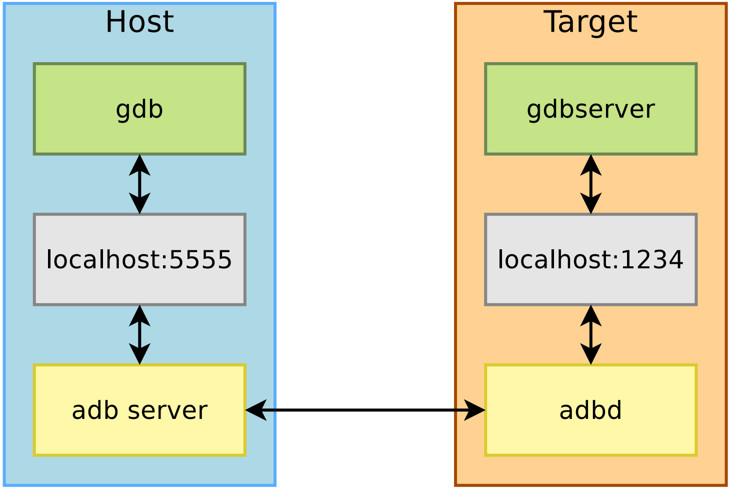
Android Debug Bridge

Use of ADB

Useful Commands

push	Copies a local file to the device
pull	Copies a remote file from the device
install	Installs the given Android package (apk) on the device
uninstall	Uninstalls the given package name from the device
lolcat	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
reboot	Reboots the device. `bootloader` and `recovery` arguments are available to select the operation mode you want to reboot to.

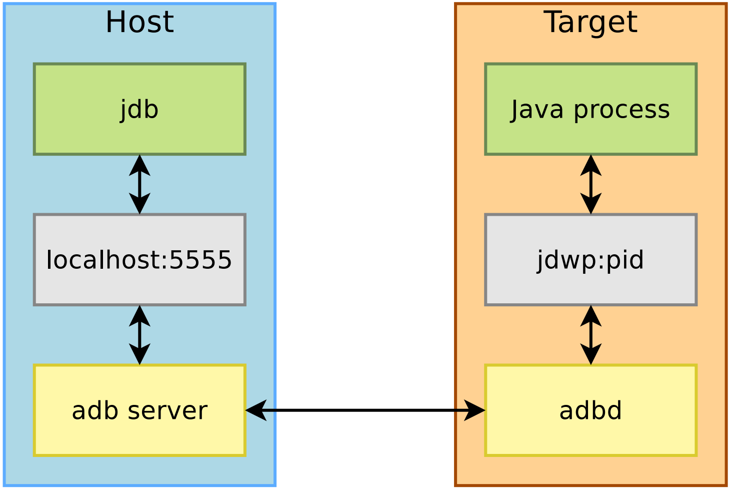
ADB forward and gdb



```
adb forward tcp:5555 tcp:1234
```

See also `gdbclient`

ADB forward and jdb



```
adb forward tcp:5555 jdwp:4242
```

Various commands

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

Android Application Development

Maxime Ripard
maxime.ripard@bootlin.com

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

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

Activity Lifecycle 1/3

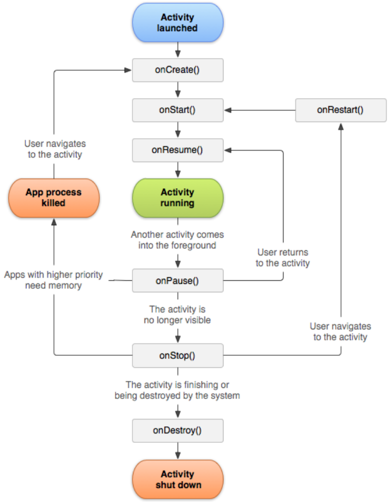
- As there is no single entry point and as the system manages the activities, activities have to define callbacks that the system can call at some point in time
- Activities can be in one of the three states on Android

Running	The activity is on the foreground and has focus
Paused	The activity is still visible on the screen but no longer has focus. It can be destroyed by the system under very heavy memory pressure
Stopped	The activity is no longer visible on the screen. It can be killed at any time by the system

Activity Lifecycle 2/3

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

Activity Lifecycle 3/3


Activity launched
onCreate()
onStart()
onRestart()
onResume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

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

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

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
  - 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

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.

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.

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 life cycle 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.

Android Application Development

Resources

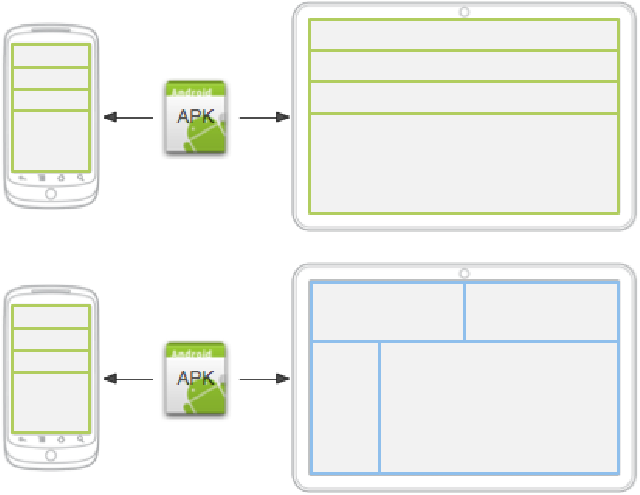
Applications Resources

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

Resources Directory

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

Resources

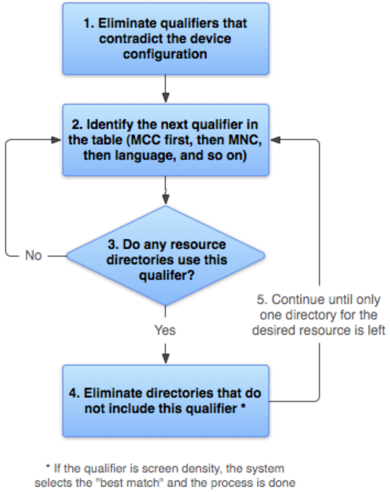


Credits: `http://developer.android.com`

Alternative Resources

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

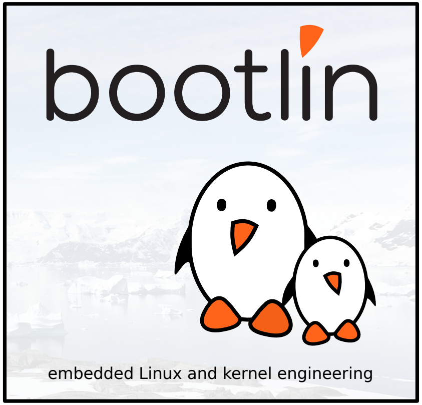
Resources Selection


1. Eliminate qualifiers that contradict the device configuration
2. Identify the next qualifier in the table (MCC first, then MNC, then language, and so on)
3. Do any resource directories use this qualifer?
No
Yes
4. Eliminate directories that do not include this qualifier *
5. Continue until only one directory for the desired resource is left
* If the qualifier is screen density, the system selects the "best match" and the process is done

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

Demo Time!

Maxime Ripard
maxime.ripard@bootlin.com



Hardware used in this demo

Using DevKit8000 boards from Embest

- OMAP3530 SoC from Texas Instruments
- 256 MB RAM, 256 MB flash
- 4"3 TFT LCD touchscreen
- 1 USB 2.0 host, 1 USB device
- 100 Mbit Ethernet port
- DVI-D / HDMI display connector
- Expansion port, JTAG port, etc.
- Currently sold in Europe at 269 EUR (V.A.T. not included) by NeoMore.