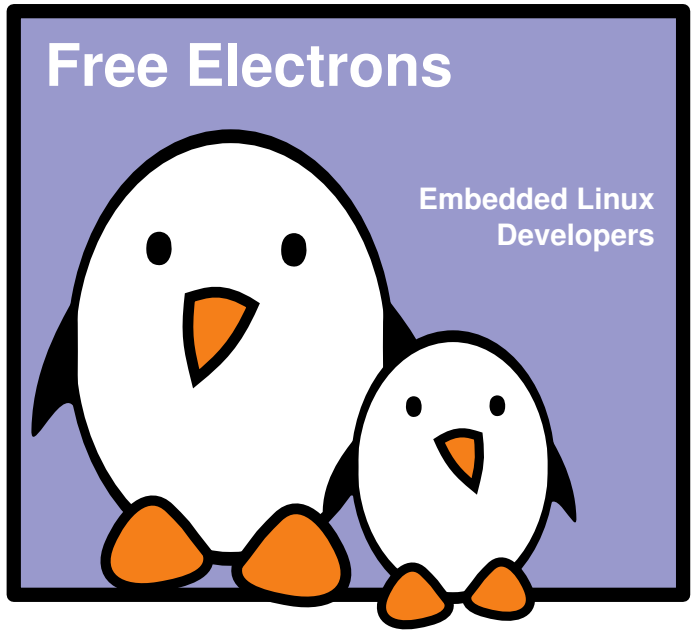
Android System Development



Maxime Ripard
Free Electrons
maxime.ripard@free-electrons.com

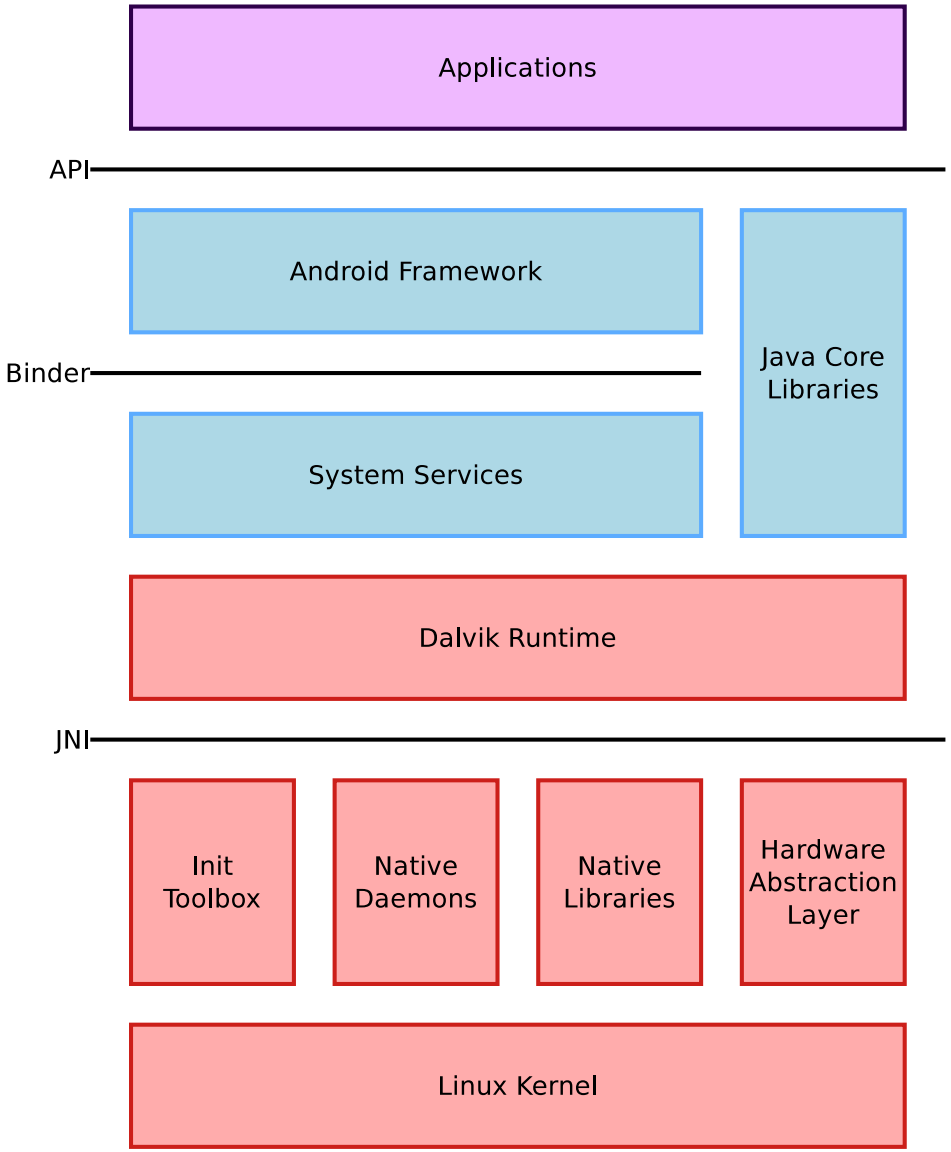
Maxime Ripard

- Embedded Linux engineer and trainer at Free Electrons 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.
  - `http://www.free-electrons.com`
- Contributor to various open-source projects: Barebox, Linux, Buildroot
- Living in Toulouse, south west of France

Why Modify Android ?

- Because we have a custom set of functions on a given device that isn't handled by the vanilla Android
- Because we want to add a different UI/set of features than AOSP
- Because since we have access to the source code, we can!

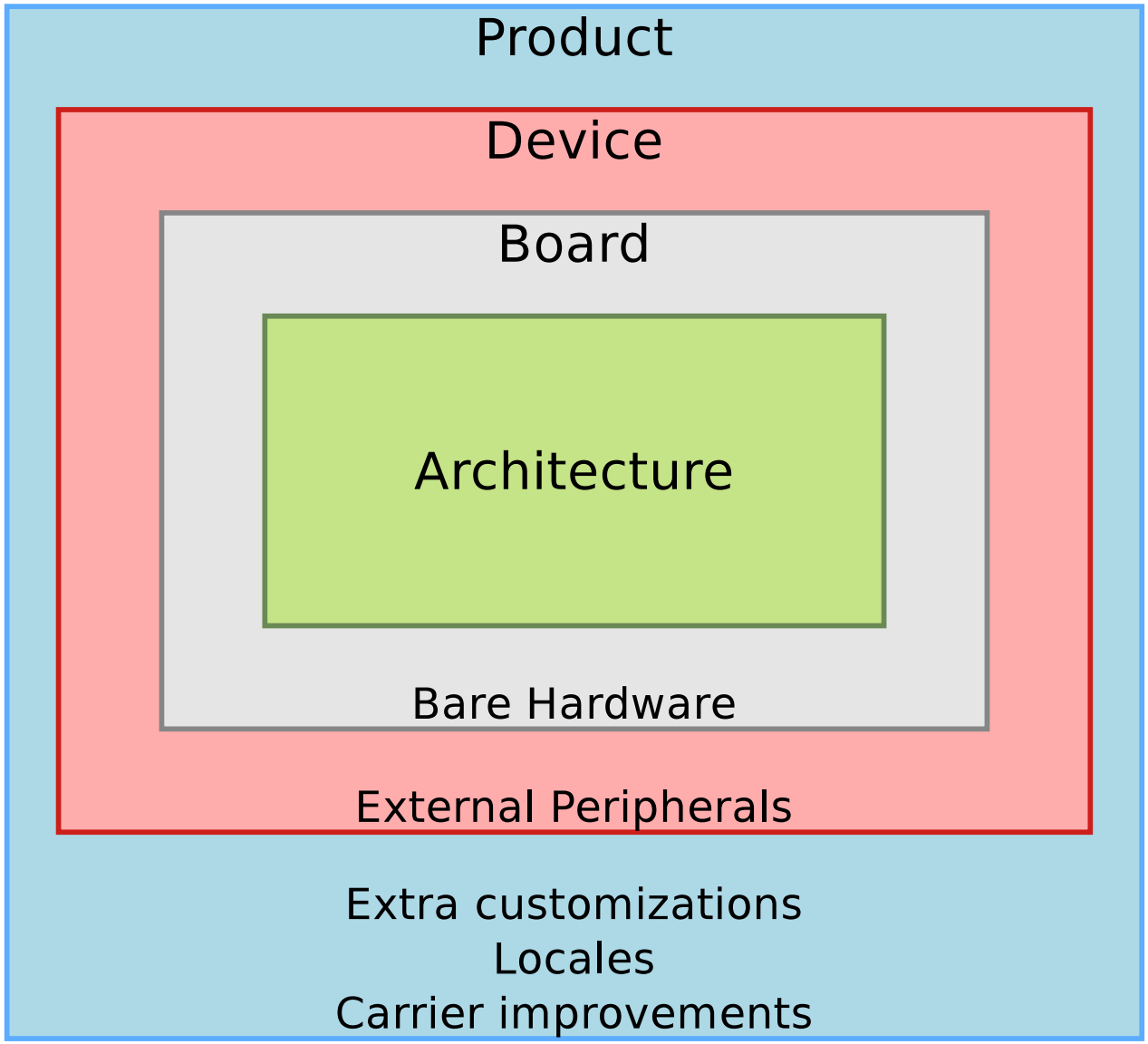
The Android Stack



Devices in the Source Tree

- The build system is using a configuration mechanism based on various layer of hardware design
- The upper layer, the product is typically the one we want to build a system for
- The build system has no other configuration mechanism: no Kconfig, text file, or any other mecanism that can be easily modified

Product, devices and boards



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`

Our New Product

- AndroidProducts.mk

```
PRODUCT_MAKEFILES += \
        $(LOCAL_DIR)/full_product.mk
```

- full_product.mk

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

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

- vendorsetup.sh

```
add_lunch_combo full_product-eng
```

Add a New Component

- The integration of new components into the build system are done through Android.mk files
- These files just contains where the source code is and what the expected result should be: an executable, a shared library, a jar file, etc.
- All the build magic is abstracted by the build system

Add a New Library

- Create a new subfolder under the `device/company/product/lib` folder
- This folder will contain the source code in itself, plus the headers and the `Android.mk` file
- To include a new package into a given build, you need to add the package name to the content of the `PRODUCT_PACKAGES` variable
- Let's use the libusb as an example

Steps to add the libusb

- The first step is obviously to untar the source code in the `device/company/product/lib/libusb` folder
- Since the Android build system doesn't rely at all on external build tools, we have some bits that are missing
- The first one is that the source code is missing the `config.h` header file, that is usually generated with the `autoconf` tool, through the `./configure` command
- When we try to compile, it lacks two things: the `TIMESPEC_TO_TIMEVAL` macro and the `timerfd.h` header
- The first one should be declared by the c library, let's add the macro declaration where the source code is using it
- The `timerfd.h` file is one of the header exposed by the Linux Kernel. It is not in the headers found in the toolchain used by Android though, so we will need to disable the use of such feature through the command `./configure --disable-timerfd`

Library Makefile

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

LOCAL_SRC_FILES := \
        core.c \
        descriptor.c \
        io.c \
        sync.c \
        os/linux_usbfs.c

LOCAL_C_INCLUDES += \
        $(LOCAL_PATH)/os

LOCAL_MODULE:= libusb
LOCAL_MODULE_TAGS:= optional
LOCAL_PRELINK_MODULE:= false

include $(BUILD_SHARED_LIBRARY)
```

Next steps

- The libusb uses the device files under the folder `/dev/bus/usb`, that are only readable by the root user and the `usb` group.
- The device files creations are handled by the `init` application, so that when a new device appears on the system, its associated device files will be created as well
- The specific part of `init` that handles these files is called `ueventd`
- It uses a set of rules to create device files with the correct permissions at runtime
- If we want to use a binary running as user and that has access to the USB devices, we'll have to modify these permissions

Modify the permissions of the device files

- Permissions on the files are the enforced thanks to two components:
  - Permissions on files on the system are defined in a C header: `android_filesystem_config.h`
  - Permissions for device files are handled by `ueventd` that uses a file called `ueventd.rc`
- We can modify the permissions on the device files by adding or modifying a `ueventd.rc` file

```
/dev/bus/usb/*    0666    root      usb
```

Integrate a binary that uses the libusb

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

LOCAL_SRC_FILES:= usb_bin.c

LOCAL_C_INCLUDES += \
    $(JNI_H_INCLUDE) \
    device/company/device/lib/libusb

LOCAL_SHARED_LIBRARIES := libusb

LOCAL_MODULE := usbbin
LOCAL_MODULE_TAGS := optional
LOCAL_PRELINK_MODULE := false

include $(BUILD_EXECUTABLE)
```

JNI Bindings

- 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

Libusb bindings

- Create a new subfolder under `device/company/device/framework/USBFramework/jni`
- Write the JNI bindings so that the Java can call functions from the libusb

Libusb bindings

```
static struct libusb_device_handle *devh;

JNIEXPORT void JNICALL Java_com_fe_USB_init(JNIEnv *env,
                                            jobject obj)
{
ret = libusb_init(NULL);
if (ret < 0)
                return ret;

        devh = libusb_open_device_with_vid_pid(NULL,
                                               0xdead,
                                               0xbeef);
        if (!devh) {
                libusb_exit(NULL);
                return -EINVAL;
        }

        return 0;
}
```

Android.mk for the bindings

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

LOCAL_SRC_FILES:= \
    usb_jni.c

LOCAL_C_INCLUDES += \
    $(JNI_H_INCLUDE) \
    device/company/device/lib/libusb

LOCAL_SHARED_LIBRARIES := \
    libusb

LOCAL_MODULE := libusb_jni
LOCAL_MODULE_TAGS := optional
LOCAL_PRELINK_MODULE := false

include $(BUILD_SHARED_LIBRARY)
```

Java Framework

```
package com.fe;

class USB {
    private native void init();

    public USB() {
        init();
    }

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

Android.mk for the framework

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

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

LOCAL_MODULE_TAGS := optional

LOCAL_MODULE:= com.fe.usb

include $(BUILD_JAVA_LIBRARY)
```

Permissions file

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

Applications using that code

- Any applications can now use the part of the framework we just added, using a regular Java call
- However, by default, the jar file generated won't be loaded by dalvik, resulting in a error at runtime.
- You need to add extra bits to the manifest file so that both the Play Store and Dalvik knows that you need the given jar file to work properly
- You need to add the `<uses-library>` XML tag

Android Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
      xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.fe.usb">

    <application
          android:icon="@drawable/icon"
          android:label="@string/app_name">

...

        <uses-library android:name="com.fe.usb"
                      android:required="true" />
    </application>
</manifest>
```

Questions?

Maxime Ripard

maxime.ripard@free-electrons.com