Forking Android considered harmful

Chris Simmonds
Stefan Lengfeld (inovex GmbH)

Agenda

- The AOSP developer’s dilemma
- Overview to making changes of the AOSP
- Recap of possible options (1-4)
- Why is forking such a bad idea?
- A proposal to avoid forking, based on ...
- Yocto-style layers applied to AOSP
- Q&A

Chris Simmonds

▸ LinkedIn

▸ e-mail

▸ website

▸ github

Android and Linux Embedded Developer

- Professional geek
- Working with embedded Linux since 1999
- Android since 2009
- Author of *Mastering Embedded Linux Programming*
- Speaker at many conferences and workshops
- Organiser of the AOSP and AAOS Meetup

Main topics of Interest:

- The point where software meets hardware
- Linux, Yocto, AOSP
- Open source
- Building better communities

Stefan Lengfeld

- LinkedIn
- e-mail
- website
- github

Android and Linux Embedded Developer

- since 2017 with inovex
- since 2014 a professional embedded software developer
- many more years a Linux enthusiast

Main topics of Interest:

- Embedded Systems (Linux and Android)
- Linux Kernel
- Build Systems
- Linux Graphics Stack
- Performance Analysis

The AOSP developer’s dilemma

Porting Android to a new platform always requires some changes to the upstream AOSP

Typical reasons to make changes are:

- board support
- custom HALs
- custom services
- custom libraries - native and jar
- custom system apps, maybe including a custom launcher
- modified system behaviour - e.g. "kiosk" mode app

How to best make those changes?

Overview to making changes of the AOSP

General two different things to look at:

- (Fork) Patching existing code in the AOSP source tree
- (Addition) Adding code (= repositories) to the AOSP source tree

The methods: The good, the bad, the ugly

- Option 1: Using a copy script
- Option 2: Patching on top
- Option 3: Extending with a localmanifest
- Option 4: Copying and extending the upstream manifest

Option 1: Using a copy script

The downstream project ...

- maintains zip archive or git repository with some copied and modified files
- writes a README to specify the upstream manifest and version from Google or BSP vendor
- has script with cp or rsync to overwrite files in the AOSP tree

How to checkout ...

```
repo init # from upstream XML manifest by Google or BSP Vendor
repo sync -j4
git clone # one repo from the downstream project or a zip file
cd project
./copy_to_aosp.sh
```

Examples: profusion/android-sdk-addon-example some customers

Option 2: Patching on top

The downstream project ...

- maintains a git repository with patches
- writes a README to specify the upstream manifest and version from Google or BSP vendor
- has a script that applies the patches to the ASOP source tree

How to checkout ...

```
repo init # from upstream XML manifest by Google or BSP Vendor
repo sync -j4
git clone # one repo from the downstream project
cd project
./apply_patches.sh
```

Examples: remote-android/redroid-patches
GloDroidCommunity/pine64-pinephone aospandaaos/vim3-aaos

Interlude 1: repo manifest trick - remove-project

Repo documentation: repo docs/manifest-format.md

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

  <!-- Adding some repositories for cool-device -->
  <project path="device/cool-device" name="cool-device-deivce" remote="cool-project" />
  <project path="packages/apps/cool-device" name="cool-device-app"
      remote="cool-project" />

  <!-- Forking some repositories for cool-device-->
  <remove-project name="packages/apps/Launcher3" />
  <project path="packages/apps/Launcher3" name="cool-device-launcher3"
      remote="cool-project" />

</manifest>
```

Option 3: Extending with a localmanifest

The downstream project ...

- maintains a git repository for the local manifest XML file
- writes a README to specify the upstream manifest and version from Google or BSP vendor
- adds and forks some repositories

How to checkout ...

```
repo init # from upstream XML manifest by Google or BSP vendor
git clone <downstream manifest repo> .repo/local_manifests
repo sync -j4
```

Examples: raspberry-vanilla/android_local_manifest
snappautomotive/firmware-local_manifest

Interlude 2: repo manifest trick - include

Repo documentation: repo docs/manifest-format.md

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

  <!-- Copy of the upstream Manifest by Google -->
  <include name="aosp/android-13.0.0_r6.xml"/>

  <!-- Additions from cool-project -->
  <include name="cool-device.xml"/>

</manifest>
```

And the file "cool-device.xml" contains the same XML tags as seen above.

Option 4: Copying and extending the upstream manifest

The downstream project ...

- maintains it's own manifest repository for repo
- copies and includes upstream XML manifest files from Google or a BSP vendor
- adds and forks some repositories

How to checkout ...

```
repo init # from downstream XML manifest by the project
repo sync -j4
```

Examples: baylibre/ti projectceladon/manifest CalyxOS/platform_manifest

Forking is bad because...

- it adds a maintenance burden: you have to back port security and bug fixes, or ...
- you have to resolve conflicts when you merge in a later version from upstream
- often, the changes made in the fork are mixed together: it's hard to tell wich changes are for board support, which for system apps, etc
- the overhead of eventually migrating to a later version is a barrier, leading to devices running outdated code e.g. android 9 and kernel 4.19

To summarise: forking = technical debt

This is especially true for a code base as large and activly developed as AOSP

How to avoid forking

Fundamental rule

don’t change upstream AOSP code, including the manifest

Instead

- add code by adding repo projects via a local manifest
- patch upstream code in a managed way

Additional benefit if you group the patches so that they are orthogonal to each other, making if possible to combine patches from multiple sources

So, who does it this way?

Lesson 1: Glodroid

Glodroid (`https://glodroid.github.io/`) is an Android distro for various boards including PinePhone, PinePhone-Pro, and Raspberry Pi 4

Early versions of Glodroid made changes to the manifest, as described earlier

Glodroid 2.0 switches to a "mono-repository" approach using patches for core system and for board support

Glodroid mono-repository

Stated advantages (from `https://github.com/GloDroid/glodroid_manifest`)

- No need to maintain forks. All necessary delta is stored in the form of patches. Benefits: Such patches are a lot easier to maintain. No need to merge/rebase fork repositories. Patches are much easier to review by external auditors to ensure high project trustworthiness.
- Atomic changes. Benefits: Classic approach may require synchronization of multiple changes in different repositories. Gerrit has a "Topic: " field for this purpose, but GitHub/GitLab doesn't support such flow. The mono-repository approach makes it possible. CI is much easier to integrate.
- Decouple devices or devices group from each other. As time has shown, maintaining all devices under the same manifest is impractical. It significantly increases the release cycle since all devices must be validated before publishing. Also, some devices may require custom patches on top of AOSP or vendor components, while others don't. It also allows different people to maintain different devices independently, benefiting from using a common code, without waiting for each other.

Lesson 2: Yocto Project

Yocto Project is a general purpose tool for building large code bases and creating image files ready to be flashed into device memory - just like the AOSP build system

Mapping Yocto concepts onto AOSP

Yocto	AOSP	Description
bitbake	soong	the build tool
Recipe	Android.bp	Instructions to build a module
bitbake classes	`build/make/core/*`	the logic to build the images

Yocto meta layers

A **meta layer** is a group of related meta data, which may include

- recipes
- bitbake classes
- config files

There are several types of layer

- Machine (BSP), e.g. meta-raspberrypi
- Distribution, e.g. meta-agl-core (Automotive Grade Linux)
- Software, e.g. meta-qt6 (Recipes for QT6 graphics libraries)

You combine layers together to create a product

This is a very powerful concept

The official list of layers can be found at `https://layers.openembedded.org/layerindex/`

Proosal: add layers on top of AOSP

This would be a wrapper round AOSP: we don’t fork it in any way

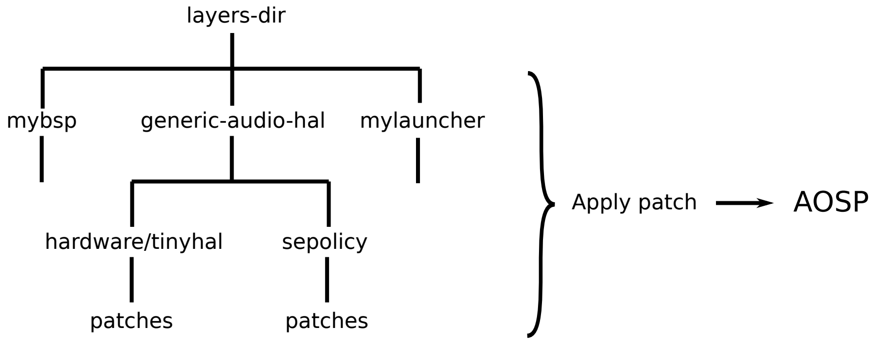
An AOSP layer = local manifests, patches, plus meta data describing the layer

Multiple layers can be combined to create the result you want

Layers have different purposes, including

- Board support, e.g. for Raspberry Pi
- HAL
- build system changes, e.g. to create images in the right format
- framework changes
- add system apps and services

AOSP layers, diagramatically



Making a product out of layers

Layers are the building blocks

To make a product, you need to group layers together

- could be a simple json file that lists the layers
- and a set of tools that can process it

Tooling required

For the developer

- tools to create layers
- tools to group layers into a product

For the builder

- tools to get a product

Example: getting a product

The workflow to get the source code for a product would be:

- read the product file
- use repo init to create a local AOSP repository
- get local manifests
- get layers and check for incompatabilities
- repo sync
- apply patches

Conclusions

- Forking the whole of AOSP for each product is not a good idea
- Using local manifests solves some problems, but not all
- We need a better solution, maybe based on the meta layers concept of Yocto Project
- We need your help

- Questions?