Android, System Approach

Ça va bien se passer

Alizée Penel

GISTRE, EPITA

v2018.1

Android Build System : Basics

Goals

Interests

Build systems are designed to meet several goals:

- Integrate all the software components into a workspace and a working image
- Be able to easily reproduce a build

Actually, they build software using the existing building system shipped within each component.

Several solutions: Yocto, Buildroot, ...

Android solution

Originally, 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
- In AOSP sources, everything in the `build` directory

Makefile

- All makefile rules are defined in `build/make`
- Non recursive makefiles
- Incremental builds take a lot of time
- File extension: `.mk`
- Android module makefile : `Android.mk`

Since Nougat

As building an Android is a pain in the a**, Google decides to rework it.

New build components were integrated into AOSP :

- ninja.
- kati

Ninja first goal is to build things quickly.

Maintainers *did not* convert all Android makefiles into ninja files. They implemented `kati` which converts makefiles into ninja files.

It takes 10 secondes to convert makefiles but takes 1-2 minutes to rereads all of them.

Not enough

Even if ninja builds faster than make, makefile loading is still horrible.

A new weapon : soong

- File extension : `.bp`
- AOSP main manifest : `Android.bp` at the root of the sources
- Syntax intentionally similar to `Bazel`
- It takes only 5 secondes to regenerate the main manifest in instead of 1-2 minutes for an Android.mk.

Future

Google plans to abandon make but not tomorrow:

- Still 3593 Android,mk in Oreo sources
- Only 1415 soong files.

AOSP Building

Basically, Google recommends to use an Ubuntu distribution.

All information on Android website, so RTFM !

Docker

Greatest way to build any Android version
You still need to install some packages on your host in order to be able to launch the emulator

Build commands

Official documentation

Requirements

Supports only bash. Do not try any other shell. You will be disappointed.

Build

```
source build/envsetup.sh
lunch
make -j42
```

Clean

```
make clean
```

envsetup.sh

Purpose

It adds many useful shell environment variables and commands to the current environment.

These macros will serve several purposes:

- Configure and setup the build system
- Ease the navigation in the source code
- Ease the development process

Exported environment variables

- `ANDROID_BUILD_PATHS`: path to all the folders containing build tools
- `ANDROID_PRODUCT_OUT`: path to the compiled target directory
- `OUT`: alias to `$ANDROID_PRODUCT_OUT`
- `JAVA_HOME`: path to Java environment

Defined shell commands

lunch	Used to configure the build system by choosing a target
printconfig	Prints the current build configuration
croot	Changes the directory to the top of the source tree
cproj	Changes the directory to the top of the current package
godir	Go to the directory containing the given file
m	Makes the whole build from any directory in the source tree
mm(a)	Builds all the modules defined in the current directory (and their dependencies)
mmm(a)	Builds all the modules defined in the given directory (and their dependencies)
{c,gg,j,res,man,sep,s,rc}grep	Greps the given pattern on all the {C/C++, Gradle, Java, res/*.xml, AndroidManifest.xml, sepolicy, source, *.rc} files
hmm	List all the commandes given by sourcing envsetup.sh

Build System Configuration

Configuration

The Android build system is not much configurable compared to other build systems, but it is possible to modify to some extent.

You can:

- choose what product you want to build,
- add extra flags for the C compiler,
- have a given package built with debug options,
- specify the output directory, ...

This is done either through the `lunch` command or through a `buildspec.mk` file

lunch

`lunch` is a shell function defined in `build/envsetup.sh`

It is the easiest way to configure a build.

Without any argument, it will ask to choose among a list of known "combos", or launch it with the desired combos as argument.

It sets the environment variables needed for the build.

You can declare new combos through the `add_lunch_combo command`

Combo definition

`<product name>-<build variant>` *e.g: full_fugu-userdebug*

Check your configuration

```
printconfig
```

Exported target environment variables

- `TARGET_PRODUCT`: Which product to build
- `TARGET_BUILD_VARIANT`: Which set of modules to build
- `TARGET_BUILD_TYPE`: either `release` or `debug`. If debug is set, it will enable some debug options across the whole system.

Existing build variants

- user : Includes modules tagged `optional`
- userdebug : Includes modules tagged `optional` or `debug` (strace)
- eng : Includes modules tagged `optional`, `debug` or `eng` (e2fsprogs)

Makefile variables

- `HOST_ARCH`: `x86` or `x86_64`
- `HOST_OS`: Generally linux
- `HOST_BUILD_TYPE`: Build properties for the host, either `release` or `debug`.
- `BUILD_ID`:
  - Specify the branch name and/or a release candidate.
  - It must be a single word, and is capitalized by convention.

buildspec.mk

If you have only one product or you want to do more fine-grained configuration, `buildspec.mk` file is here for that.

Place it at the top of the sources, and it will be used by the build system to get its configuration instead of relying on the environment variables.

It offers more variables to modify:

- compiling a given module with debugging symbols,
- add C compiler flags,
- change the output directory, etc.

A sample is available in `build/make/buildspec.mk.default`, with lots of comments on the various variables.

Compilation

Build commands

There are many build commands:

```
make
make droid            # normal build

make showcommands     # build in verbose mode

make all              # builds everything, whether it is included in
                      # the product definition or not

make services         # builds system server (Java) and friends
```

Build commands


```
make modules          # list all the modules available in the build
                      # system

make <module>         # builds only the module

make sdk              # builds the complete SDK package

mm           # Builds all the modules in the current directory

mmm <directory>       # Builds all the modules in the given directory
```

Useful documentation on elinux.org

Results

Output

All the output is generated in the `out/` directory, outside of the source code directory

This directory contains mostly two subdirectories:

- `host/`
- `target/`

These directories contain all the objects files compiled during the build process.

It generates the system images in the `out/target/product/<device_name>/` directory

These images are:

boot.img	A basic Android image, containing only the needed components to boot: a kernel image and a minimal system
system.img	The remaining parts of Android. Much bigger, it contains most of the framework, applications and daemons
userdata.img	A partition that will hold the user generated content. Mostly empty at compilation.
recovery.img	A recovery image that allows to be able to debug or restore the system when something nasty happened.

Never ever

Do not use these recovery images. Use twrp.

Android boot images

The boot images are actually an Android-specific format, that holds most of what the bootloader expects

They contains useful information:

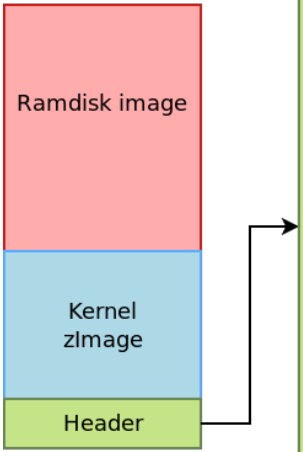
- the kernel command line
- where to load the kernel
- the kernel image,
- an optional initramfs

A custom `mkbootimg` tool is used by Android to generate these images at compilation time from the kernel and the system it's generating.

We can tweak the behaviour of that tool from the build system configuration, that allows a great flexibility

Hacker tool

abootimg




```
struct boot_img_hdr {
    unsigned char magic[8];/* "ANDROID!" */
    unsigned kernel_size;  /* size in bytes */
    unsigned kernel_addr;  /* physical load addr */

    unsigned ramdisk_size; /* size in bytes */
    unsigned ramdisk_addr; /* physical load addr */

    unsigned second_size;  /* size in bytes */
    unsigned second_addr;  /* physical load addr */

    unsigned tags_addr;    /* physical addr for kernel tags */
    unsigned page_size;    /* flash page size we assume, usually 2048 */
    unsigned unused[2];    /* future expansion: should be 0 */

    unsigned char name[16]; /* asciiz product name */
    unsigned char cmdline[512];

    unsigned id[8]; /* timestamp / checksum / sha1 / etc */
};

from system/core/mkbootimg/bootimg.h
```

Cleaning

Cleaning commands


```
male clobber          # cleans all the files generated by previous
make clean            # compilations

make clean-<module>   # removes all the files generated by the
                      # compilation of the given module

make installclean     # removes the installed files for the current
                      # combo. Usefull if you work with several
                      # products (avoid a full rebuild each time you
                      # change from one to the other).
```

Build System : Advanced

Android modules

Definition

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.

Example


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

LOCAL_SRC_FILES = hello_world.c
LOCAL_MODULE = HelloWorld

LOCAL_MODULE_TAGS = optional
include $(BUILD_EXECUTABLE)
```

Module variable definitions

`LOCAL_PATH` Tells the build system where the current module is

`include $(CLEAR_VARS)` Cleans the previously declared

`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

Example

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

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

include $(BUILD_PREBUILT)
```

Every module variable is prefixed by `LOCAL_*`.

The list of the variables cleared is in `build/make/core/clear_vars.mk`.

`LOCAL_CFLAGS` Extra C compiler flags to use to build the module

`LOCAL_SHARED_LIBRARIES` List of shared libraries this module depends on at compilation time

`LOCAL_PACKAGE_NAME` Equivalent to LOCAL_MODULE for Android packages

`LOCAL_C_INCLUDES` List of paths to extra headers used by this module

`LOCAL_REQUIRED_MODULES` Express that a given module depends on another at runtime, and therefore should be included in the image as well

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 `optional`
  - Installs non-packaged modules that have no tags specified
  - `ro.secure = 1`
  - `ro.debuggable = 0`
  - ADB is disabled by default
- `userdebug` is `user`, except:
  - Also installs modules tagged with `debug`
  - `ro.debuggable = 1`
  - ADB is enabled by default

Tags

- eng is userdebug, plus
  - Installs modules tagged as eng and development
  - `ro.secure = 0`
  - `ro.kernel.android.checkjni = 1`

Android module tags

`LOCAL_MODULE_TAGS` can take many tags, separated by whitespace :

`user`	Not allowed anymore in build system (since Lollipop).
`optional`	Replace `user` tag, include module in each build.
`debug`	Include module in `userdebug` builds.
`eng`	Include module in `eng` builds.
`tests`	Declare module as a test package `make tests dist`
`samples`	Declare module as a sample, never included

List of build targets

`BUILD_EXECUTABLE` Builds a normal ELF binary to be run on the target

`BUILD_JAVA_LIBRARY` Builds a Java library (.jar) to be used on the target

`BUILD_RAW_EXECUTABLE` Builds a binary to be run on bare metal

`BUILD_STATIC_JAVA_LIBRARY` Builds a static Java library to be used on the target

`BUILD_HOST_EXECUTABLE` Builds an ELF binary to be run on the host

`BUILD_HOST_JAVA_LIBRARY` Builds a Java library to be used on the host

`BUILD_HOST_STATIC_LIBRARY` Builds a static library for the host

`BUILD_HOST_SHARED_LIBRARY` Builds a shared library for the host

List of build targets

`BUILD_STATIC_LIBRARY` Builds a static library for the target

`BUILD_SHARED_LIBRARY` Builds a shared library for the target

`BUILD_RAW_STATIC_LIBRARY` Builds a static library to be used on bare metal

`BUILD_PREBUILT` Used to install prebuilt files on the target (configuration files, kernel)

`BUILD_HOST_PREBUILT` Used to install prebuilt files on the host

`BUILD_PACKAGE` Builds a standard Android package (.apk)

The complete list is available in `buid/core/config.mk`.

Useful make macros

In the `build/make/core/definitions.mk` file, you will find useful macros to use in the `Android.mk` file, that mostly allows you to:

- Find files: `all-makefiles-under`, `all-subdir-c-files`, etc.
- Transform them: `transform-c-to-o`, etc.
- Copy them: `copy-file-to-target`, etc.
- And some utilities: `my-dir`, `inherit-package`, etc.

All these macros should be called through Make's `call` command, possibly with arguments.

Want to create your own macro ?

Check in build/make/core/definition.mk first !

Building and cleaning modules

The files generated will be put in `out/target/product/$TARGET_DEVICE/obj/<module_type>/<module_name>_intermediates`.

make modules or make <module>

It won't regenerate a new image. Just useful to make sure that modules build.

make

It will build your module but it will not be in the result image if it is tagged as `optional`.
Add the module name to the `PRODUCT_PACKAGES` variable to integrate it the final image.

Android products

Add a new product

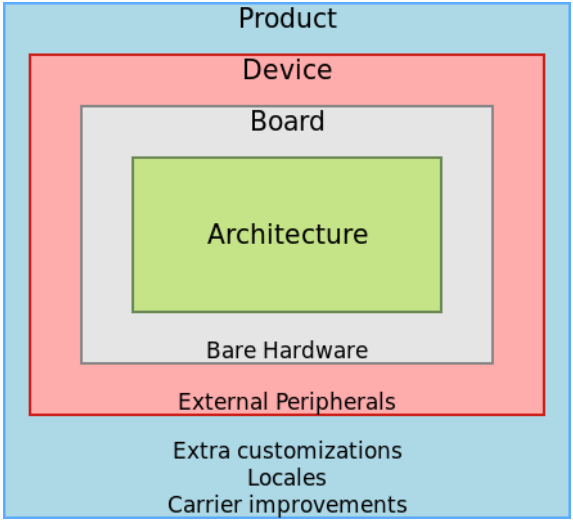
Android build system allows to build multiple devices with the same source tree.

All the product definitions should be put in `device/<company>/<device name>`.

Add a new product in `lunch`

Create a `vendorsetup.sh` file in the device directory, with the right calls to `add_lunch_combo`.

Products, devices and boards



AndroidProducts.mk

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.

```
PRODUCT_MAKEFILES := \
    $(LOCAL_DIR)/full_toto.mk
```

full_<device>.mk

```
$(call inherit-product, device/<company>/<device>/device.mk)
$(call inherit-product, $(SRC_TARGET_DIR)/product/generic.mk)

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

device.mk


```
PRODUCT_PACKAGES += FooBar

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

DEVICE_PACKAGE_OVERLAYS := device/mybrand/mydevice/overlay
```

Overlays

This is a mechanism used by products to override resources already defined in the source tree, without modifying the original code.

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.

BoardConfig.mk

You will also need a `BoardConfig.mk` file along with the product definition.

While the product definition was mostly about the build system in itself, the board definition is more about the hardware.

However, this is poorly documented and sometimes ambiguous so you will probably have to dig into the `build/make/core/Makefile` at some point to see what a given variable does.

Minimal BoardConfig.mk


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

Some boards variables

`TARGET_ARCH_VARIANT` Variant of the selected architecture

`TARGET_EXTRA_CFLAGS` Extra C compiler flags to use during the whole build

`TARGET_CPU_SMP` Does the CPU have multiple cores?

`TARGET_USERIMAGES_USE_EXT4` Use ext4 as filesystem for our generated partitions

`BOARD_SYSTEMIMAGE_PARTITION_SIZE` Size of the system partitions in bytes

`BOARD_NAND_PAGE_SIZE` For NAND flash, size of the pages as given by the datasheet

`TARGET_NO_RECOVERY` Do not build the recovery image

`BOARD_KERNEL_CMDLINE` Boot arguments of the kernel commandline

Kernels

Basics

In the source tree

Android is a pure userspace software stack.

The build system isn't designed to build the kernel.

However, AOSP integrate precompiled kernels:

- Device kernels are located in `device/<company>/<device name>-kernel`
- Emulator kernels are located in `prebuilts/qemu-kernels`

Kernel integration : BoardConfig.mk


```
BOARD_KERNEL_BASE        := 0x00000000
BOARD_KERNEL_PAGESIZE    := 4096
BOARD_KERNEL_TAGS_OFFSET := 0x01E00000
BOARD_RAMDISK_OFFSET     := 0x02000000

BOARD_KERNEL_CMDLINE := console=ttyHSL0,115200,n8 androidboot.hardware=bullhead \
                        boot_cpus=0-5
BOARD_KERNEL_CMDLINE += lpm_levels.sleep_disabled=1 msm_poweroff.download_mode=0
BOARD_KERNEL_CMDLINE += loop.max_part=7
```

Optionally

```
BOARD_MKBOOTIMG_ARGS := --ramdisk_offset $(BOARD_RAMDISK_OFFSET) --tags_offset \
                        $(BOARD_KERNEL_TAGS_OFFSET)
TARGET_BOARD_KERNEL_HEADERS := device/google/marlin/kernel-headers
```

Kernel integration : device.mk


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

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

Kernel sources

AOSP has a kernel repository per device.

The complete list is here

Nowdays, a vanilla kernel for x86 architecture with Android features activated works. No need to use goldfish repository.

Clone must be aside AOSP source tree.

Toolchains

- Goggle toolchains (`prebuilts/gcc`)
- Linaro's toolchains (for ARM only)

Kernel build

```
export ARCH=arm64
export CROSS_COMPILE=aarch64-linux-android-
cd hikey-linaro
git checkout -b android-hikey-linaro-4.1 origin/android-hikey-linaro-4.1
make hikey_defconfig
make
```

At the directory root, you will find `build.config.<arch>`.

Source it, run make.

To avoid the copy at each kernel new build, you can:

```
export TARGET_PREBUILT_KERNEL=<your_kernel_path>/arch/arm/boot/zImage-dtb
```

Before Oreo, Android did not support kernel modules : modules statically bundled in the kernel binary.

Thanks to Greg Kroah-Hartman help, Google did a great work about Android kernel supports.

Documentation

BoardConfig.mk

```
vendor_lkm_dir := device/$(vendor)/lkm-4.x
BOARD_VENDOR_KERNEL_MODULES := \
  $(vendor_lkm_dir)/vendor_module_a.ko \
  $(vendor_lkm_dir)/vendor_module_b.ko \
  $(vendor_lkm_dir)/vendor_module_c.ko
```

Overlays can be use.

Android kernel development

Sources

Google provides a kernel source tree in order to help OEM: kernel/common

Configuration

Specific Android kernel configurations are located in another repository : here

Android defconfig generation:

```
ARCH=<arch> scripts/kconfig/merge_config.sh <.../>/device_defconfig \
                                            <.../>/android-base.cfg \
                        <.../>/android-base-<arch>.cfg \
                        <.../>/android-recommended.cfg
```