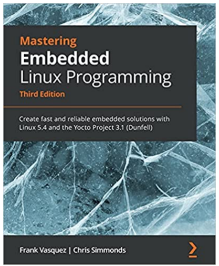
The AOSP Build System

Chris Simmonds

Embedded Linux Conference Europe 2023

About Chris Simmonds



- Consultant and trainer
- Author of *Mastering Embedded Linux Programming*
- Working with embedded Linux since 1999
- Android since 2009
- Speaker at many conferences and workshops

"Looking after the Inner Penguin" blog at https://2net.co.uk/

Mastodon: @csimmonds@fosstodon.org https://fosstodon.org/@csimmonds

https://uk.linkedin.com/in/chrisdsimmonds/

Agenda

- The AOSP build system
- Soong
- Kati
- Ninja
- Bazel

The AOSP build system

- AOSP build system is similar to others (OpenEmbedded, Buildroot), with its own peculiarities
- AOSP is a huge project (>50 MLOC of C++, Rust, Java, Kotlin)
- The AOSP build system has evolved over time into something quite unique
- This talk is based on Android 13 - it will probably all change in Android 14 as part of the migration to Bazel (maybe I'll come back next year to talk about that :-))

Note: it does not include building a Linux kernel, bootloader, or any other ancillary binaries. It's up to the you (or the SoC vendor) to piece it all together (resulting in some truly weird stuff)

Getting AOSP: repo

- AOSP is a collection of git repositories (> 1100 in T/13)
- First, get a manifest listing the git trees to clone, optionally specifying a version tag with `-b`:

```
$ repo init -u https://android.googlesource.com/platform/manifest -b android-13.0.0_r35
```

- Then clone the git trees
  - you get ALL the packages in one go (150GB), rather downloading on demand as in OpenEmbedded

```
$ repo sync
```

Selecting and building a target product

- Set up the shell environment

```
$ source build/envsetup.sh
```

- Select the target using lunch (a shell function defined in envsetup.sh)

```
$ lunch aosp_cf_x86_64_phone-userdebug
```

- Each target is defined by an `AndroidProduct.mk` e.g. `aosp_cf_x86_64_phone-userdebug` comes from `device/google/cuttlefish/AndroidProduct.mk`:

```
COMMON_LUNCH_CHOICES := aosp_cf_x86_64_phone-userdebug
```

- Start the build:

```
$ m
```

- Then have a coffee, have another coffee, take a vacation, ...

Outputs

- Each product lists the Android modules to build in Makefile variable `PRODUCT_PACKAGES`

```
PRODUCT_PACKAGES += CuttlefishService vsoc_input_service
```

- ... which you can dump using `get_build_var`:

```
$ get_build_var PRODUCT_PACKAGES
[...] sample_camera_extensions.xml CuttlefishService vsoc_input_service e2fsck [...]
```

- The final results are image files in `out/target/product/[device name]`

```
$ cd out/target/product/vsoc_x86_64
$ ls *.img
boot.img
ramdisk.img
super.img
system.img
vendor.img
[...]
```

- Typically you flash these to the device using fastboot

Recipes

- Android modules are defined in recipes in one of two formats
- Android.bp
  - written in blueprint, introduced in O/8
  - T/13 has > 8000 Android.bp files
- Android.mk
  - written in Makefile format
  - deprecated, but still hanging around
  - in T/13 there are about 1000

The build tools

There are three main tools:

- **soong**: parses Android.bp files and generates ninja manifests (and some makefile fragments)
- **kati**: parses Android.mk and all the other makefile fragments and generates more ninja manifests
- **ninja**: parses the ninja manifests, generates the dependency tree for the target to built and schedules jobs

Maybe I can make it simpler ...

Kati

Genetically-engineered Augment,
follower of Khan Noonien Singh

Kati

Kahn Noonian Singh

Nothing to do this this story

Kati

Kahn Noonian Singh

Nothing to do this this story

Kati

Dr Noonian Soong

AI genius, inventor of Data

Kati

Soong

Data is an android but not
"Android (tm)" - as far as we know

Kati

Soong

Also not part of this story

Kati

Soong

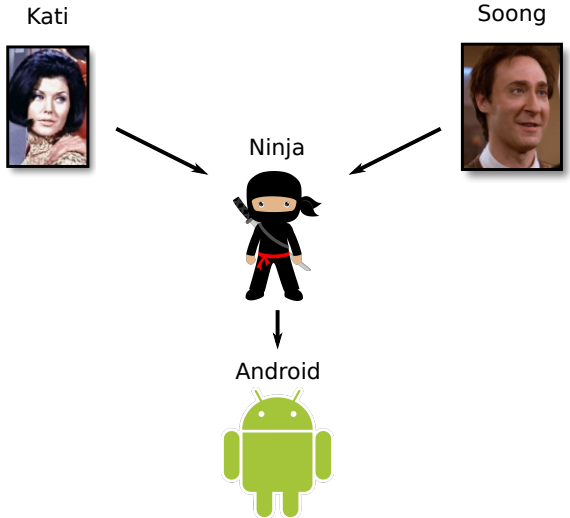
Ninja

Japanese assassin

Kati

Ninja

Soong



How did we get here?

- 2008 C/1.5: GNU Make: single Makefile composed at build time from fragments (*.mk) (ref: Recursive Make Considered Harmful) But, Make does not scale well: slow to start up, even if there is no work to do; no progress indication
- 2016 N/7: kati and ninja

  Kati implements the logic built into makefiles and outputs a dependency tree as a ninja manifest

  Ninja schedules jobs to reach the goal - showing progress
- 2017 O/8: soong

  Soong was intended as a replacement for makefiles and Kati. New format: Blueprint

  Progress of Soong has been slow: in T/13 there are still 1000's of makefile fragments
- 2023 U/14 (probably): Bazel

  How to solve a problem with software architecture? Add another layer

- The AOSP build system
- Soong
- Kati
- Ninja
- Bazel

Soong

- Soong reads `Android.bp` files written in Blueprint language
- The Blueprint language is "JSON-like", and also similar to Bazel BUILD files
- Blueprint is declarative (no build logic)
- The build logic is implemented in soong modules, written in go
- Code: `$AOSP/build/soong`
- Doc `$AOSP/build/soong/README.md`

Starting the build

The build is started with m, mm, mmm, or make, which are implemented in `build/envsetup.sh`

```
function get_make_command()
{
    # If we're in the top of an Android tree, use soong_ui.bash instead of make
    if [ -f build/soong/soong_ui.bash ]; then
        # Always use the real make if -C is passed in
        for arg in "$@"; do
            if [[ $arg == -C* ]]; then
                echo command make
                return
            fi
        done
        echo build/soong/soong_ui.bash --make-mode
    else
        echo command make
    fi
}
function make()
{
    _wrap_build $(get_make_command "$@") "$@"
}
```

If cwd is `$AOSP`, then make == m, otherwise make == make (!)

m, mm and mmm

Builds modules in either Android.bp or Android.mk files

```
m or make          build all modules for target (default droid)
mm                 unconditionally build the module in the cwd
mmm dir1,dir2,...  unconditionally build modules in directory list
```

The `droid` target for `m` and `make` invoke all tasks needed to generate the final images and other artifacts

`mm` and `mmm` only build the Android.bp and Android.mk files listed

soong_ui

`soong_ui` is started by m and friends: it is the driver for the whole build process

Command-line options for `soong_ui`:

```
soong_ui
  --make-mode        simulate make, build a makefile target
  --dumpvar-mode     dump one makefile variable
  --dumpvars-mode    print a list of makefile variables

no wildcard allowed in variable names

Code is in build/soong/cmd/soong_ui
```

`get_build_var` is a wrapper for dumpvar mode

Help with make

Help with m targets

```
$ m help
[...]
m [<goals>]
Common goals are:

    clean                   (aka clobber) equivalent to rm -rf out/
    checkbuild              Build every module defined in the source tree
    droid                   Default target
    nothing                 Do not build anything, just parse and validate the build structure

    java                    Build all the java code in the source tree
    native                  Build all the native code in the source tree

    host                    Build all the host code (not to be run on a device) in the source tree
    target                  Build all the target code (to be run on the device) in the source tree
[...]
```

See `build/make/Usage.txt` for more info

By default it will append `-j(nproc + 2)` to `m`

Example Android.bp

A simplified version of the Android.bp for logcat

```
system/logging/logcat
|-- Android.bp
|-- logcat.cpp
```

```
cc_binary {
    name: "logcat",
    srcs: ["logcat.cpp"],
    shared_libs: ["libbase", "libprocessgroup",],
    cflags: ["-Werror"],
}
```

- The module type is `cc_binary`
- The module is called `logcat`
- Has one source file: `logcat.cpp`
- Links with libraries `libbase`, and `libprocessgroup`
- Builds an executable which will be installed into `$OUT/system/bin/logcat`

Dependencies

Implicit: libraries, for example, are automatically added as a dependency

Explicit: other dependencies are given using `required`, followed by the list of modules that this module depends on

```
    required: [ "module1", "module2",],
```

All dependencies will be build and installed into the staging area before this module is built

Blueprint modules

Examples of types of module

`cc_binary`	Native binary
`cc_library_shared`	Shared library
`cc_library_static`	Static library
`cc_binary_host`	Host binary
`cc_library_host_shared`	Host shared library
`cc_library_host_static`	Host static library
`java_library (*)`	Java library
`android_app (*)`	Android app
`prebuilt_etc (*)`	Prebuilt installed into etc
`cc_prebuilt_binary (*)`	Prebuilt installed into bin

(*) Since Q/10

Soong Modules Reference:
`m soong_docs`

The docs are generated in:
`out/soong/docs/soong_build.html`

Soong modules

Soong module types (e.g. `cc_binary` are registered like this:

```
build/soong/cc/binary.go:    ctx.RegisterModuleType("cc_binary", BinaryFactory)
```

There are approx 300 module types in T/13

Each module implements logic to build the module type (similar to bbclass in OE)

Soong outputs

In the first phase, soong parses **all** Android.bp files and writes build rules to `out/soong/build.ninja`

```
"analyzing Android.bp files and generating ninja file at out/soong/build.ninja"
```

This is a *big* file: 6 to 10 GiB

Has to be regenerated whenever any Android.bp files are added or changed

At this point, Soong does not know what the build target is, so it parses all Android.bp files, even if your target does not depend on that module

... takes a long time ...

Also writes install rules to `out/soong/Android-[product name].mk` in Android.mk format which is processed by kati later

... and dependencies to `out/soong/late-[product name].mk`

Soong log files

Generated each time soong is run

Log rotation 4? not sure out/soong*.log

TBD - describe what you can find in here ...

- The AOSP build system
- Soong
- Kati
- Ninja
- Bazel

Kati

- Kati is a GNU Make clone
  - upstream code: `https://github.com/google/kati`
  - doc: `https://github.com/google/kati/blob/master/INTERNALS.md`
- Written specifically to build AOSP
- Parses makefiles into Ninja manifests and variable lists
- Implements the logic encoded in the many Makefile functions and macros
- The binary is bundled with AOSP in `prebuilts/build-tools/linux-x86/bin/ckati`

Android.mk

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

LOCAL_MODULE := helloworld-mk
LOCAL_SRC_FILES := helloworld.c
LOCAL_VENDOR_MODULE := true
LOCAL_SHARED_LIBRARIES := liblog

include $(BUILD_EXECUTABLE)
```

- The module is called `helloworld-mk`
- Has one source file: `helloworld.c`
- Links with shared library `liblog`
- `include $(BUILD_EXECUTABLE)` brings in the build rules to build an executable which will be installed into `$OUT/system/bin/logcat`

Looks a lot like Buildroot

Dependencies

Implicit: libraires, etc, same as for Android.bp

Explicit: dependencies in Android.mk look like this

```
LOCAL_REQUIRED_MODULES += gallium_dri
```

Note that a module in an Android.mk file cannot have a dependency on a module defined in an Android.bp

Ninja outputs

Ninja generates a ninja manifest named `out/build-[device name].ninja` which contains the build rules to generate the Android modules listed in the *.mk files

e.g. `out/build-aosp_cf_x86_64_phone.ninja`

... quite a bit file ... 600 to 900 MiB

Also generates a manifest with dependencies for packaging in

`out/build-aosp_cf_x86_64_phone-package.ninja`

- The AOSP build system
- Soong
- Kati
- Ninja
- Bazel

Ninja

- Ninja reads manifests generated by Soong and Kati
- Calculates dependencies for given target
- Schedules jobs needed to reach the target
- Upstream code: `https://github.com/google/kati`
- Ninja 1.9.0 is bundled in AOSP 13 in `prebuilts/build-tools/linux-x86/bin/ninja`

Ninja syntax

Here is a basic ninja manifest file (taken from https://ninja-build.org/manual.html

```
cflags = -Wall

rule cc
  command = gcc $cflags -c $in -o $out

build foo.o: cc foo.c
```

The main elements are:

Variables: e.g. `cflags`, dereference using `$cflags`

Rules: a short name for a command line, e.g. `cc`

Build statements: declare a relationship between input and output (i.e. `build outputs: rule inputs` shows how to generate the output when needed)

Note that variables `$in` and `$out` are derived from the build statement

Running Ninja

Ninja is started from `soong_ui` (see `out/soong.log` for exact command line)

```
prebuilts/build-tools/linux-x86/bin/ninja droid -j 16 -f out/combined-aosp_cf_x86_64_phone.ninja
```

Note target is `droid` and `-j 16` is the parameter I passed to `m`

`out/combined-aosp_cf_x86_64_phone.ninja` brings together all the manifest generated by soong and kati:

```
builddir = out
pool highmem_pool
 depth = 2
subninja out/build-aosp_cf_x86_64_phone.ninja
subninja out/build-aosp_cf_x86_64_phone-package.ninja
subninja out/soong/build.ninja
```

Note: `subninja` includes another .ninja file; the subninja can read and modify variables from the parent manifest but changes are not seem in the parent scope

Running ninja directly

Ninja has some useful tools

```
$ prebuilts/build-tools/linux-x86/bin/ninja -t list
ninja subtools:
    browse  browse dependency graph in a web browser
     clean  clean built files
  commands  list all commands required to rebuild given targets
      deps  show dependencies stored in the deps log
     graph  output graphviz dot file for targets
    inputs  show all (recursive) inputs for a target
      path  find dependency path between two targets
     paths  find all dependency paths between two targets
     query  show inputs/outputs for a path
   targets  list targets by their rule or depth in the DAG
    compdb  dump JSON compilation database to stdout
 recompact  recompacts ninja-internal data structures
```

Now that we know how soong starts ninja, we can use tools to find useful stuff, e.g. dependencies (next slide)

Dependencies

Show dependencies for logcat

```
$ prebuilts/build-tools/linux-x86/bin/ninja -f out/combined-aosp_cf_x86_64_phone.ninja \
-t query out/target/product/vsoc_x86_64/system/bin/logcat
out/target/product/vsoc_x86_64/system/bin/logcat:
  input: rule202069
    out/soong/.intermediates/system/logging/logcat/logcat/android_x86_64_silvermont/logcat
    out/target/product/vsoc_x86_64/obj/EXECUTABLES/logcat_intermediates/logcat
    || out/target/product/vsoc_x86_64/system/lib64/libprocessgroup.so
    || out/target/product/vsoc_x86_64/system/lib64/libcgrouprc.so
[...]
  outputs:
    out/target/product/vsoc_x86_64/system/bin/dumpstate
    out/target/product/vsoc_x86_64/system/etc/init/dumpstate.rc
    device_logcat_all_targets
    out/target/product/vsoc_x86_64/obj/PACKAGING/systemimage_intermediates/system.img
[...]
```

`input:` lists the input dependencies for logcat

`output:` lists the things that depend on logcat

For example, dumpstate has a dependency on logcat (see `frameworks/native/cmds/dumpstate/Android.bp`)

Dependency graph 1/2

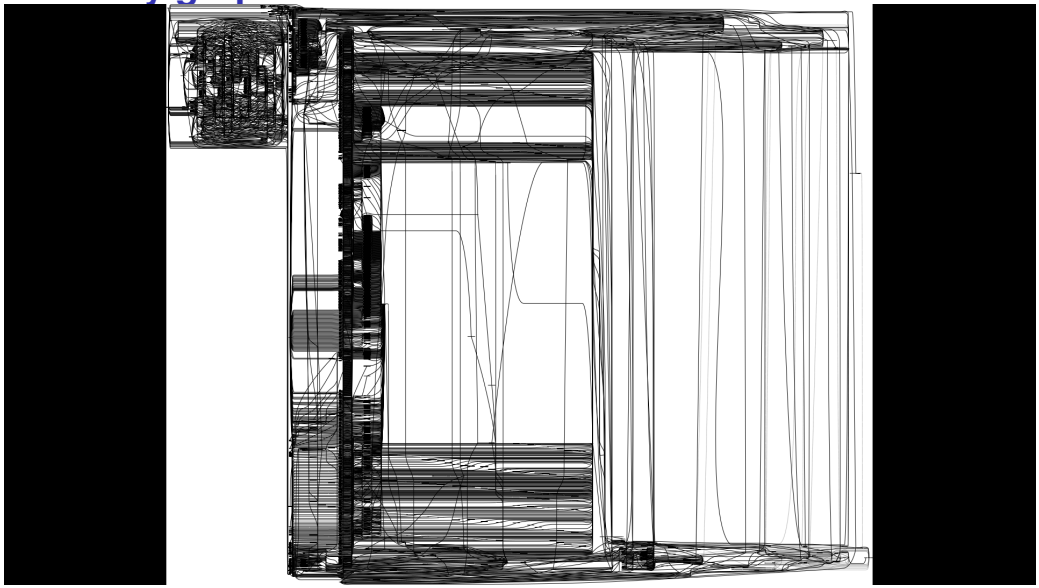
You can use the graph tool in ninja to create a graph (may take a few minutes)

```
$ prebuilts/build-tools/linux-x86/bin/ninja -f out/combined-marvin.ninja \
-t graph out/target/product/marvin/system/bin/logcat > ninja-logcat-deps.dot

$ dot -Tpdf -Nshape=box -o ninja-logcat-deps.pdf ninja-logcat-deps.dot
```

Not as helpful as I had hoped

Dependency graph 2/2



- The AOSP build system
- Soong
- Kati
- Ninja
- Bazel

The shape of things to come?

- How to solve a problem with software architecture? Add another layer
- Intention to replace Kati, Soong, and Ninja with Bazel, starting with U/14
- Currently (in T/13) only kernel build uses Bazel
- For more information about how AOSP will adapt to Bazel, see
  - "Welcome Android Open Source Project (AOSP) to the Bazel ecosystem" `https://developers.googleblog.com/2020/11/welcome-android-open-source-project.html`
  - `build/bazel/docs/concepts.md`
  - `build/bazel/docs/internal_concepts.md`

Bazel in one slide

WORKSPACE

A WORKSPACE file defines the top level of a project. For AOSP we have `$AOSP/WORKSPACE -> build/bazel/bazel.WORKSPACE`. Contains some global configuration

BUILD

Each module is defined in a BUILD file (similar to Android.bp or Android.mk)

Bazel rules (.bzl)

The logic is implemented in .bzl files. For example `$AOSP/bazel/rules/cc/cc_binary.bzl` contains the logic to build a C/C++ binary

Starlark BUILD and .bzl files are written in a which is is a Python-like called Starlark. (properly known as the "Build Language", though it is often simply referred to as "Starlark")

Bazel migration

In T/13, there is support to build the Android Common Kernel entirely in Bazel

In U/14, Bazel will replace ninja

Questions?

Slides at
https://2net.co.uk/slides/elc/aosp-build-eoss-2023.pdf

@2net_software

https://uk.linkedin.com/in/chrisdsimmonds/