Soong Build System

Before the Android 7.0 release, Android used GNU Make exclusively to describe and execute its build rules. The Make build system is widely supported and used, but at Android's scale became slow, error prone, unscalable, and difficult to test. The Soong build system provides the flexibility required for Android builds.

For this reason, platform developers are expected to switch from Make and adopt Soong as soon as possible. Send questions to the android-building Google Group to receive support.

[What is Soong?]

The Soong build system was introduced in Android 7.0 (Nougat) to replace Make. It leverages the Kati GNU Make clone tool and Ninja build system component to speed up builds of Android.

See the Android Make Build System description in the Android Open Source Project (AOSP) for general instructions and Build System Changes for Android.mk Writers to learn about modifications needed to adapt from Make to Soong.

See the build-related entries in the glossary for definitions of key terms and the Soong reference files for complete details.

Caution: Until Android fully converts from Make to Soong, the Make product configuration must specify the `PRODUCT_SOONG_NAMESPACES` value. See the Namespace modules section for instructions.

[Make and Soong comparison]

Here is a comparison of Make configuration with Soong accomplishing the same in a Soong configuration (Blueprint or `.bp`) file.

[Make example]

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

include $(CLEAR_VARS)
LOCAL_MODULE := libxmlrpc++
LOCAL_MODULE_HOST_OS := linux

LOCAL_RTTI_FLAG := -frtti
LOCAL_CPPFLAGS := -Wall -Werror -fexceptions
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/src

LOCAL_SRC_FILES := $(call \
     all-cpp-files-under,src)
include $(BUILD_SHARED_LIBRARY)
```

[Soong example]

```
cc_library_shared {
     name: “libxmlrpc++”,

     rtti: true,
     cppflags: [
           “-Wall”,
           “-Werror”,
           “-fexceptions”,
     ],
     export_include_dirs: [“src”],
     srcs: [“src/**/*.cpp”],

     target: {
           darwin: {
                enabled: false,
           },
     },
}
```

See Simple Build Configuration for test-specific Soong configuration examples.

[Android.bp file format]

By design, `Android.bp` files are simple. They don't contain conditionals or control flow statements; all complexity is handled by build logic written in Go. When possible, the syntax and semantics of `Android.bp` files are similar to Bazel BUILD files.

[Modules]

A module in an `Android.bp` file starts with a module type followed by a set of properties in `name: "value",` format:

```
cc_binary {
    name: "gzip",
    srcs: ["src/test/minigzip.c"],
    shared_libs: ["libz"],
    stl: "none",
}
```

Every module must have a `name` property, and the value must be unique across all `Android.bp` files, except for the `name` property values in namespaces and prebuilt modules, which may repeat.

The `srcs` property specifies the source files used to build the module, as a list of strings. You can reference the output of other modules that produce source files, like `genrule` or `filegroup`, by using the module reference syntax `":<module-name>"`.

For a list of valid module types and their properties, see the Soong Modules Reference.

[Types]

Variables and properties are strongly typed, with variables dynamically based on the first assignment, and properties set statically by the module type. The supported types are:

- Booleans (`true` or `false`)
- Integers (`int`)
- Strings (`"string"`)
- Lists of strings (`["string1", "string2"]`)ranslate="no" dir="ltr"})

Maps may contain values of any type, including nested maps. Lists and maps may have trailing commas after the last value.

[Globs]

Properties that take a list of files, such as `srcs`, can also take glob patterns. Glob patterns can contain the normal UNIX wildcard `*`, for example `*.java`. Glob patterns can also contain a single `**` wildcard as a path element, which matches zero or more path elements. For example, `java/**/*.java` matches both the `java/Main.java` and `java/com/android/Main.java` patterns.

[Variables]

An `Android.bp` file may contain top-level variable assignments:

```
gzip_srcs = ["src/test/minigzip.c"],
cc_binary {
    name: "gzip",
    srcs: gzip_srcs,
    shared_libs: ["libz"],
    stl: "none",
}
```

Variables are scoped to the remainder of the file they are declared in, as well as any child Blueprint files. Variables are immutable with one exception: they can be appended to with a `+=` assignment, but only before they've been referenced.

[Comments]

`Android.bp` files can contain C-style multiline `/* */` and C++ style single-line `//` comments.

[Operators]

Strings, lists of strings, and maps can be appended using the + operator. Integers can be summed up using the `+` operator. Appending a map produces the union of keys in both maps, appending the values of any keys that are present in both maps.

[Conditionals]

Soong doesn't support conditionals in `Android.bp` files. Instead, complexity in build rules that would require conditionals are handled in Go, where high-level language features can be used, and implicit dependencies introduced by conditionals can be tracked. Most conditionals are converted to a map property, where one of the values in the map is selected and appended to the top-level properties.

For example, to support architecture-specific files:

```
cc_library {
    ...
    srcs: ["generic.cpp"],
    arch: {
        arm: {
            srcs: ["arm.cpp"],
        },
        x86: {
            srcs: ["x86.cpp"],
        },
    },
}
```

[Formatter]

Soong includes a canonical formatter for Blueprint files, similar to gofmt. To recursively reformat all `Android.bp` files in the current directory, run:

```
bpfmt -w .
```

The canonical format includes four-space indents, new lines after every element of a multielement list, and a trailing comma in lists and maps.

[Special modules]

Some special module groups have unique characteristics.

[Defaults modules]

A defaults module can be used to repeat the same properties in multiple modules. For example:

```
cc_defaults {
    name: "gzip_defaults",
    shared_libs: ["libz"],
    stl: "none",
}

cc_binary {
    name: "gzip",
    defaults: ["gzip_defaults"],
    srcs: ["src/test/minigzip.c"],
}
```

[Prebuilt modules]

Some prebuilt module types allow a module to have the same name as its source-based counterparts. For example, there can be a `cc_prebuilt_binary` named `foo` when there's already a `cc_binary` with the same name. This gives developers the flexibility to choose which version to include in their final product. If a build configuration contains both versions, the `prefer` flag value in the prebuilt module definition dictates which version has priority. Note that some prebuilt modules have names that don't start with `prebuilt`, such as `android_app_import`.

[Namespace modules]

Until Android fully converts from Make to Soong, the Make product configuration must specify a `PRODUCT_SOONG_NAMESPACES` value. Its value should be a space-separated list of namespaces that Soong exports to Make to be built by the `m` command. After Android's conversion to Soong is complete, the details of enabling namespaces could change.

Soong provides the ability for modules in different directories to specify the same name, as long as each module is declared within a separate namespace. A namespace can be declared like this:

```
soong_namespace {
    imports: ["path/to/otherNamespace1", "path/to/otherNamespace2"],
}
```

Note that a namespace doesn't have a name property; its path is automatically assigned as its name.

Each Soong module is assigned a namespace based on its location in the tree. Each Soong module is considered to be in the namespace defined by the `soong_namespace` found in an `Android.bp` file in the current directory or closest ancestor directory. If no such `soong_namespace` module is found, the module is considered to be in the implicit root namespace.

Here's an example: Soong attempts to resolve dependency D declared by module M in namespace N that imports namespaces I1, I2, I3...

1. Then if D is a fully qualified name of the form `//namespace:module`, only the specified namespace is searched for the specified module name.
2. Otherwise, Soong first looks for a module named D declared in namespace N.
3. If that module doesn't exist, Soong looks for a module named D in namespaces I1, I2, I3...
4. Lastly, Soong looks in the root namespace.