Effective in 2026, to align with our trunk stable development model and ensure platform stability for the ecosystem, we will publish source code to AOSP in Q2 and Q4. For building and contributing to AOSP, use `android-latest-release`. The `android-latest-release` manifest branch will always reference the most recent release pushed to AOSP. For more information, see Changes to AOSP (/docs/whatsnew/site-updates#aosp-changes).

# Add a new device

Use the information in this page to create the makefiles for your device and product.

**Note:** This information applies only to creating a new device type and is intended specifically for company build and product teams.

Each new Android module must have a configuration file to direct the build system with module metadata, compile-time dependencies, and packaging instructions. Android uses the Soong build system (https://android.googlesource.com/platform/build/soong/+/refs/heads/android17-release/README.md). See Building Android (/docs/setup/build/building) for more information about the Android build system.

## Understand build layers

The build hierarchy includes the abstraction layers that correspond to the physical makeup of a device. These layers are described in the table below. Each layer relates to the one above it in a one-to-many relationship. For example, an architecture can have more than one board and each board can have more than one product. You may define an element in a given layer as a specialization of an element in the same layer, which eliminates copying and simplifies maintenance.

| Layer | Example | Description |
| --- | --- | --- |

| | | |
|---|---|---|
| Product | myProduct, myProduct_eu, myProduct_eu_fr, j2, sdk | The product layer defines the feature specification of a shipping product such as the modules to build, locales supported, and configuration for various locales. In other words, this is the *name* of the overall product. Product-specific variables are defined in product definition makefiles. A product can inherit from other product definitions, which simplifies maintenance. A common method is to create a base product that contains features that apply to all products, then create product variants based on that base product. For example, two products that differ only by their radios (CDMA versus GSM) can inherit from the same base product that doesn't define a radio. |
| Board/device | marlin, blueline, coral | The board/device layer represents the physical layer of plastic on the device (that is, the industrial design of the device). This layer also represents the bare schematics of a product. These include the peripherals on the board and their configuration. The names used are merely codes for different board/device configurations. |
| Arch | arm, x86, arm64, x86_64 | The architecture layer describes the processor configuration and application binary interface (ABI) running on the board. |

## Use build variants

When building for a particular product, it's useful to have minor variations on the final release build. In a module definition, the module can specify tags with `LOCAL_MODULE_TAGS`, which can be one or more values of `optional` (default), `debug`, and `eng`.

If a module doesn't specify a tag (by `LOCAL_MODULE_TAGS`), its tag defaults to `optional`. An optional module is installed only if it's required by the product configuration with `PRODUCT_PACKAGES`.

These are the currently defined build variants.

| Variant | Description |
|---|---|
| `eng` | This is the default flavor.<br>• Installs modules tagged with `eng` or `debug`.<br>• Installs modules according to the product definition files, in addition to tagged modules.<br>• `ro.secure=0`<br>• `ro.debuggable=1` |

- `ro.kernel.android.checkjni=1`
- `adb` is enabled by default.

---

`user` The variant intended to be the final release bits.

- Installs modules tagged with `user`.
- Installs modules according to the product definition files, in addition to tagged modules.
- `ro.secure=1`
- `ro.debuggable=0`
- `adb` is disabled by default.

---

`userdebug` The same as `user`, with these exceptions:

- Also installs modules tagged with `debug`.
- `ro.debuggable=1`
- `adb` is enabled by default.

---

## Guidelines for userdebug

Running userdebug builds in testing helps device developers understand the performance and power of in-development releases. To maintain consistency between user and userdebug builds, and to achieve reliable metrics in builds used for debugging, device developers should follow these guidelines:

- userdebug is defined as a user build with root access enabled, except:
  - userdebug-only apps that are run only on-demand by the user
  - Operations that run only during idle maintenance (on charger/fully charged), such as using `dex2oatd` versus `dex2oat` for background compiles
- Don't include features that are enabled/disabled by default based on the build type. Developers are discouraged from using any form of logging that affects battery life, such as debug logging or heap dumping.
- Any debugging features that are enabled by default in userdebug should be clearly defined and shared with all developers working on the project. You should enable debugging features only on a limited-time basis until the issue you're trying to debug is resolved.

## Customize the build with resource overlays

The Android build system uses resource overlays to customize a product at build time. Resource overlays specify resource files that are applied on top of the defaults. To use resource overlays, modify the project buildfile to set `PRODUCT_PACKAGE_OVERLAYS` to a path relative to your top-level directory. That path becomes a shadow root searched along with the current root when the build system searches for resources.

The most commonly customized settings are contained in the file frameworks/base/core/res/res/values/config.xml (https://android.googlesource.com/platform/frameworks/base/+/android17-release/core/res/res/values/config.xml)
.

To set up a resource overlay on this file, add the overlay directory to the project buildfile using one of the following:

```
PRODUCT_PACKAGE_OVERLAYS := device/device-implementer/device-name/overlay
```

or

```
PRODUCT_PACKAGE_OVERLAYS := vendor/vendor-name/overlay
```

Then, add an overlay file to the directory, for example:

```
vendor/foobar/overlay/frameworks/base/core/res/res/values/config.xml
```

Any strings or string arrays found in the overlay `config.xml` file replace those found in the original file.

## Build a product

You can organize the source files for your device in many different ways. Here's a brief description of one way to organize a Pixel implementation.

Pixel is implemented with a main device configuration named `marlin`. From this device configuration, a product is created with a product definition makefile that declares product-specific information about the device such as the name and model. You can view the `device/google/marlin` directory to see how all of this is set up.

## Write product makefiles

The following steps describe how to set up product makefiles in a way similar to that of the Pixel product line:

1. Create a `device/<company-name>/<device-name>` directory for your product. For example, `device/google/marlin`. This directory will contain source code for your device along with the makefiles to build them.
2. Create a `device.mk` makefile that declares the files and modules needed for the device. For an example, see `device/google/marlin/device-marlin.mk`.
3. Create a product definition makefile to create a specific product based on the device. The following makefile is taken from `device/google/marlin/aosp_marlin.mk` as an example. Notice that the product inherits from the `device/google/marlin/device-marlin.mk` and `vendor/google/marlin/device-vendor-marlin.mk` files through the makefile while also declaring the product-specific information such as name, brand, and model.

```
# Inherit from the common Open Source product configuration
$(call inherit-product, $(SRC_TARGET_DIR)/product/core_64_bit.mk)
$(call inherit-product, $(SRC_TARGET_DIR)/product/aosp_base_telephony.mk)

PRODUCT_NAME := aosp_marlin
PRODUCT_DEVICE := marlin
PRODUCT_BRAND := Android
PRODUCT_MODEL := AOSP on msm8996
PRODUCT_MANUFACTURER := Google
PRODUCT_RESTRICT_VENDOR_FILES := true

PRODUCT_COPY_FILES += device/google/marlin/fstab.common:$(TARGET_COPY_OUT_VENDOR

$(call inherit-product, device/google/marlin/device-marlin.mk)
$(call inherit-product-if-exists, vendor/google_devices/marlin/device-vendor-mar

PRODUCT_PACKAGES += \
```

```
    Launcher3QuickStep \
    WallpaperPicker
```

See Setting product definition variables (#prod-def) for additional product-specific variables that you can add to your makefiles.

4. Create an `AndroidProducts.mk` file that points to the product's makefiles. In this example, only the product definition makefile is needed. The example below is from `device/google/marlin/AndroidProducts.mk` (which contains both marlin, the Pixel, and sailfish, the Pixel XL, which shared most configuration):

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

COMMON_LUNCH_CHOICES := \
        aosp_marlin-userdebug \
        aosp_sailfish-userdebug
```

5. Create a `BoardConfig.mk` makefile that contains board-specific configurations. For an example, see `device/google/marlin/BoardConfig.mk`.
6. *For Android 9 and lower **only***, create a `vendorsetup.sh` file to add your product (a "lunch combo") to the build along with a build variant (#build-variants) separated by a dash. For example:

```
add_lunch_combo <product-name>-userdebug
```

7. At this point, you can create more product variants based on the same device.

## Set product definition variables

Product-specific variables are defined in the product's makefile. The table shows some of the variables maintained in a product definition file.

| Variable | Description | Example |
| --- | --- | --- |
| `PRODUCT_AAPT_CONFIG` | `aapt` configurations to use when creating packages. | |
| `PRODUCT_BRAND` | The brand (for example, carrier) the software is customized for. | |
| `PRODUCT_CHARACTERISTICS` | `aapt` characteristics to allow adding variant-specific resources to a package. | `tablet`, `nosdcard` |
| `PRODUCT_COPY_FILES` | List of words like `source_path:destination_path`. The file at the source path should be copied to the destination path when building this product. The rules for the copy steps are defined in `config/makefile`. | |
| `PRODUCT_DEVICE` | Name of the industrial design. This is also the board name, and the build system uses it to locate `BoardConfig.mk`. | `tuna` |
| `PRODUCT_LOCALES` | A space-separated list of two-letter language code, two-letter country code pairs that describe several settings for the user, such as the UI language and time, date, and currency formatting. The first locale listed in `PRODUCT_LOCALES` is used as the product's default locale. | `en_GB`, `de_DE`, `es_ES`, `fr_CA` |
| `PRODUCT_MANUFACTURER` | Name of the manufacturer. | `acme` |
| `PRODUCT_MODEL` | End-user-visible name for the end product. | |
| `PRODUCT_NAME` | End-user-visible name for the overall product. Appears in the **Settings > About** screen. | |
| `PRODUCT_OTA_PUBLIC_KEYS` | List of over-the-air (OTA) public keys for the product. | |
| `PRODUCT_PACKAGES` | List of the APKs and modules to install. | Calendar contacts |

| | | |
|---|---|---|
| `PRODUCT_PACKAGE_OVERLAYS` | Indicates whether to use default resources or add any product specific overlays. | `vendor/acme/overlay` |
| `PRODUCT_SYSTEM_PROPERTIES` | List of the system property assignments in the format `"key=value"` for the system partition. System properties for other partitions can be set via `PRODUCT_<PARTITION>_PROPERTIES` as in `PRODUCT_VENDOR_PROPERTIES` for the vendor partition. Supported partition names: `SYSTEM`, `VENDOR`, `ODM`, `SYSTEM_EXT`, and `PRODUCT`. | |

## Configure default system language and locale filter

Use this information to configure the default language and system locale filter, then enable the locale filter for a new device type.

### Properties

Configure both the default language and the system locale filter using dedicated system properties:

- `ro.product.locale`: for setting the default locale. This is initially set to the first locale in the `PRODUCT_LOCALES` (#product-locales) variable; you can override that value. (For more information, see the Setting product definition variables (#prod-def) table.)
- `ro.localization.locale_filter`: for setting a locale filter, using a regular expression applied to locale names. For example:
  - Inclusive filter: `^(de-AT|de-DE|en|uk).*` - allows only German (Austria and Germany variants), all English variants of English, and Ukrainian
  - Exclusive filter: `^(?!de-IT|es).*` - excludes German (Italy variant), and all variants of Spanish.

### Enable the locale filter

To enable the filter, set the `ro.localization.locale_filter` system property string value.

By setting the filter property value and the default language through `oem/oem.prop` during factory calibration you can configure restrictions without baking the filter into the system image. You

ensure that these properties are picked up from the OEM partition by adding them to the `PRODUCT_OEM_PROPERTIES` variable as indicated below:

```
# Delegation for OEM customization
PRODUCT_OEM_PROPERTIES += \
    ro.product.locale \
    ro.localization.locale_filter
```

Then in production the actual values are written to `oem/oem.prop`, to reflect the target requirements. With this approach, the default values are retained during the factory reset, so the initial settings look exactly like a first setup to the user.

## Set ADB_VENDOR_KEYS to connect over USB

The `ADB_VENDOR_KEYS` environment variable enables device manufacturers to access debuggable builds (-userdebug and -eng, but not -user) over adb without manual authorization. Normally adb generates a unique RSA authentication key for each client computer, which it will send to any connected device. This is the RSA key shown in the adb authorization dialog. As an alternative you can build known keys into the system image and share them with the adb client. This is useful for OS development and especially for testing because it avoids the need to manually interact with the adb authorization dialog.

To create vendor keys, one person (usually a release manager) should:

1. Generate a key pair using `adb keygen`. For Google devices, Google generates a new key pair for each new OS version.
2. Check the key pairs in, somewhere in the source tree. Google stores them in `vendor/google/security/adb/`, for example.
3. Set the build variable `PRODUCT_ADB_KEYS` to point to your key directory. Google does this by adding an `Android.mk` file in the key directory that says `PRODUCT_ADB_KEYS := $(LOCAL_PATH)/$(PLATFORM_VERSION).adb_key.pub`, which helps ensure that we remember to generate a new key pair for each OS version.

Here's the makefile Google uses in the directory where we store our checked-in key pairs for each release:

```
PRODUCT_ADB_KEYS := $(LOCAL_PATH)/$(PLATFORM_VERSION).adb_key.pub

ifeq ($(wildcard $(PRODUCT_ADB_KEYS)),)
  $(warning ========================)
  $(warning The adb key for this release)
  $(warning )
  $(warning   $(PRODUCT_ADB_KEYS))
  $(warning )
  $(warning does not exist. Most likely PLATFORM_VERSION in build/core/version_defaul
  $(warning has changed and a new adb key needs to be generated.)
  $(warning )
  $(warning Please run the following commands to create a new key:)
  $(warning )
  $(warning   make -j8 adb)
  $(warning   LOGNAME=android-eng HOSTNAME=google.com adb keygen $(patsubst %.pub,%,$
  $(warning )
  $(warning and upload/review/submit the changes)
  $(warning ========================)
  $(error done)
endif
```

To use these vendor keys, an engineer only needs to set the `ADB_VENDOR_KEYS` environment variable to point to the directory in which the key pairs are stored. This tells `adb` to try these canonical keys first, before falling back to the generated host key that requires manual authorization. When `adb` can't connect to an unauthorized device, the error message will suggest that you set `ADB_VENDOR_KEYS` if it's not already set.



Last updated 2026-06-17 UTC.