If you've never built the engine before, first see Setting up the Engine development environment.

Contents

Depending on the platform you are making changes for, you may be interested in all or only some of the sections below:

General Compilation Tips

- For local development and testing, it's generally preferable to use `--unopt` builds. These builds will have additional logging and checks enabled, and generally use build and link flags that lead to faster compilation and better debugging symbols. If you are trying to do performance testing with a local build, do not use the `--unopt` flag.
- Link Time Optimization: Optimized builds also perform Link Time Optimization of all binaries. This makes the linker take a lot of time and memory to produce binaries. If you need optimized binaries but don't want to perform LTO, add the `--no-lto` flag.
- Android and iOS expect both a `host` and `android` (or `ios`) build. It is critical to recompile the host build after upgrading the Dart SDK (e.g. via a `gclient sync` after merging up to head), since artifacts from the host build need to be version matched to artifacts in the Android/iOS build.
- Web, Desktop, and Fuchsia builds have only one build target (i.e. `host` or `fuchsia`).
- Make sure to exclude the `out` directory from any backup scripts, as many large binary artifacts are generated. This is also generally true for all of the directories outside of the `engine/src/flutter` directory.
- For Googlers: Goma (go/ma) is a distributed compiler service that can vastly speed up build times. The variables to use Goma compilation are set by default if goma-related environment variables are detected, and can be explicitly set via the `--goma/--no-goma` flag to the `flutter/tools/gn` wrapper script.
    - Flutter Engine requires Fuchsia's Goma RBE (as opposed to Chromium's `goma.chromium.org` RBE) for remote building. This is due to Flutter Engine building with the same versions of Clang that Fuchsia uses.

      The following idempotent script can be used to fetch and run a Goma client which is preconfigured to use the correct RBE for building Flutter Engine:

      ```
      #!/bin/bash

      # Customize this to where you would like Goma to be installed.
      export GOMA_DIR="$HOME/flutter_goma"

      # Download client. Assumes cipd from depot_tools is on path.
      echo 'fuchsia/third_party/goma/client/${platform}  integration' | cipd
      ```

```
ensure -ensure-file - -root "$GOMA_DIR"

# Authenticate
"$GOMA_DIR/goma_auth" login --browser

# Start Goma
GOMA_LOCAL_OUTPUT_CACHE_DIR="$GOMA_DIR/.goma_cache"
"$GOMA_DIR/goma_ctl.py" ensure_start

# On macOS, have the build access xcode through symlinks in the build
root. Required for goma builds to work.
if [ "$(uname)" == "Darwin" ]; then export
FLUTTER_GOMA_CREATE_XCODE_SYMLINKS=1; fi
```

- Goma will fail remotely if it tries to access files that reside outside of the build root. When building configurations for macOS or iOS (i.e. configurations that require an Xcode-vended SDK or toolchain) locally, you can have the build create and use symlinks by adding the `--xcode-symlinks` argument to the `flutter/tools/gn` wrapper script or `export FLUTTER_GOMA_CREATE_XCODE_SYMLINKS=1` to your bash/zsh/whatever rc.
- If you run into errors regarding too many open file handles, follow the directions in the *How to install goma* sections of go/ma.
- If you run into `compiler binary hash mismatch` errors and local fallback builds while trying to build using Goma, then Goma is most likely using an RBE that doesn't host the compiler binaries that Flutter Engine supports building with. Try setting the following environment variables to use Fuchsia's RBE and then restart Goma:

```
export GOMA_SERVER_HOST=rbe-prod1.endpoints.fuchsia-infra-goma-
prod.cloud.goog
export GOMA_SERVER_PORT=443
```

Using a pre-built Dart SDK

When targeting the host and desktop, on CI we use a pre-built Dart SDK vended by the Dart team. To use the same setup locally, define the environment variable `FLUTTER_PREBUILT_DART_SDK=1`, do a `gclient sync`, and pass the flag `--prebuilt-dart-sdk` to `//flutter/tools/gn`.

Using a custom Dart SDK

`gclient sync` downloads the Dart SDK sources in `engine/src/third_party/dart`. Files in this directory can be edited. To ensure those changes are visible, pass the flag `--no-prebuilt-dart-sdk` to `//flutter/tools/gn`.

Compiling for Android (from macOS or Linux)

These steps build the engine used by `flutter run` for Android devices.

Run the following steps, from the `src` directory created in Setting up the Engine development environment:

1. `git pull upstream main` in `src/flutter` to update the Flutter Engine repo.

2. `gclient sync` to update dependencies.

3. Prepare your build files

   - `./flutter/tools/gn --android --unoptimized` for device-side executables.
   - `./flutter/tools/gn --android --unoptimized --android-cpu=arm64` for newer 64-bit Android devices.
   - `./flutter/tools/gn --android --android-cpu x86 --unoptimized` for x86 emulators.
   - `./flutter/tools/gn --android --android-cpu x64 --unoptimized` for x64 emulators.
   - `./flutter/tools/gn --unoptimized` for host-side executables, needed to compile the code.
     - On macOS hosts, add the `--xcode-symlinks` argument when using Goma.

4. Build your executables

   - `ninja -C out/android_debug_unopt` for device-side executables.
   - `ninja -C out/android_debug_unopt_arm64` for newer 64-bit Android devices.
   - `ninja -C out/android_debug_unopt_x86` for x86 emulators.
   - `ninja -C out/android_debug_unopt_x64` for x64 emulators.
   - `ninja -C out/host_debug_unopt` for host-side executables.
   - These commands can be combined. Ex: `ninja -C out/android_debug_unopt && ninja -C out/host_debug_unopt`
   - For MacOS, you will need older version of XCode(9.4 or below) to compile android_debug_unopt and android_debug_unopt_x86. If you only care about x64, you can ignore this
   - For Googlers, consider also adding the flag `--goma` to your gn command, then using `autoninja` to parallelize the build using Goma. Before that, you may need to set up `GOMA_DIR` environmental variable, which, depending on where you install Goma, may be in `~/goma` or `depot_tools/.cipd_bin`.

This builds a debug-enabled ("unoptimized") binary configured to run Dart in checked mode ("debug"). There are other versions, see [[Flutter's modes]].

If you're going to be debugging crashes in the engine, make sure you add `android:debuggable="true"` to the `<application>` element in the `android/AndroidManifest.xml` file for the Flutter app you are using to test the engine.

See [[The flutter tool]] for instructions on how to use the `flutter` tool with a local engine. You will typically use the `android_debug_unopt` build to debug the engine on a device, and `android_debug_unopt_x64` to debug in on a simulator. Modifying dart sources in the engine will require adding a `dependency_override` section in you app's `pubspec.yaml` as detailed here.

Note that if you use particular android or ios engine build, you will need to have corresponding host build available next to it: if you use `android_debug_unopt`, you should have built `host_debug_unopt`, `android_profile` -> `host_profile`, etc. One caveat concerns cpu-flavored builds like `android_debug_unopt_x86`: you won't be able to build `host_debug_unopt_x86` as that configuration is not supported. What you are expected to do is to build `host_debug_unopt` and symlink `host_debug_unopt_x86` to it.

Compiling everything that matters on Linux

The following script will update all the builds that matter if you're developing on Linux and testing on Android and created the `.gclient` file in `~/dev/engine`:

```
set -ex

cd ~/dev/engine/src/flutter
git fetch upstream
git rebase upstream/main
gclient sync
cd ..

flutter/tools/gn --unoptimized --runtime-mode=debug
flutter/tools/gn --android --unoptimized --runtime-mode=debug
flutter/tools/gn --android --runtime-mode=profile
flutter/tools/gn --android --runtime-mode=release

cd out
find . -mindepth 1 -maxdepth 1 -type d | xargs -n 1 sh -c 'ninja -C $0 || exit 255'
```

For `--runtime-mode=profile` build, please also consider adding `--no-lto` option to the `gn` command. It will make linking much faster with a small sacrifice on the binary size and memory usage (which probably doesn't matter for debugging or performance benchmark purposes.)

Compiling for iOS (from macOS)

These steps build the engine used by `flutter run` for iOS devices.

Run the following steps, from the `src` directory created in the steps above:

1. `git pull upstream main` in `src/flutter` to update the Flutter Engine repo.
2. `gclient sync` to update dependencies.
3. `./flutter/tools/gn --ios --unoptimized` to prepare build files for device-side executables (or `--ios --simulator --unoptimized` for simulator, and if working on iPhone 4s or older, `--ios --ios-cpu=arm --unoptimized`).

- This also produces an Xcode project for working with the engine source code at `out/ios_debug_unopt`
- For a discussion on the various flags and modes, see [[Flutter's modes]].
- Add the `--xcode-symlinks` argument when using goma.

4. `./flutter/tools/gn --unoptimized` to prepare the build files for host-side executables.

- Add the `--xcode-symlinks` argument when using goma.

5. `ninja -C out/ios_debug_unopt && ninja -C out/host_debug_unopt` to build all artifacts (use `out/ios_debug_sim_unopt` for Simulator, `out/out/ios_debug_unopt_arm` for iPhone 4s or older).
    - For Googlers, consider also using the `--goma` flag with gn, then building with `autoninja` to parallelize the build using Goma.

See [[The flutter tool]] for instructions on how to use the `flutter` tool with a local engine. You will typically use the `ios_debug_unopt` build to debug the engine on a device, and `ios_debug_sim_unopt` to debug in on a simulator. Modifying dart sources in the engine will require adding a `dependency_override` section in you app's `pubspec.yaml` as detailed here.

See also instructions for debugging the engine in a Flutter app in Xcode.

Compiling for macOS or Linux

These steps build the desktop embedding, and the engine used by `flutter test` on a host workstation.

1. `git pull upstream main` in `src/flutter` to update the Flutter Engine repo.

2. `gclient sync` to update your dependencies.

3. `./flutter/tools/gn --unoptimized` to prepare your build files.

    - `--unoptimized` disables C++ compiler optimizations. On macOS, binaries are emitted unstripped; on Linux, unstripped binaries are emitted to an `exe.unstripped` subdirectory of the build.

- Add the `--xcode-symlinks` argument when using goma on macOS.

4. `ninja -C out/host_debug_unopt` to build a desktop unoptimized binary.
    - If you skipped `--unoptimized`, use `ninja -C out/host_debug` instead.
    - For Googlers, consider also using the `--goma` flag with gn, then building with `autoninja` to parallelize the build using Goma.

See [[The flutter tool]] for instructions on how to use the `flutter` tool with a local engine. You will typically use the `host_debug_unopt` build in this setup. Modifying dart sources in the engine will require adding a `dependency_override` section in you app's `pubspec.yaml` as detailed here.

Compiling for Windows

You can only build selected binaries on Windows (mainly `gen_snapshot` and the desktop embedding).

On Windows, ensure that the engine checkout is not deeply nested. This avoid the issue of the build scripts working with excessively long paths.

1. Make sure you have Visual Studio installed (non-Googlers only). Debugging Tools for Windows 10 must be installed.

2. `git pull upstream main` in `src/flutter` to update the Flutter Engine repo.

3. Ensure long path support is enabled on your machine. Launch PowerShell as an administrator and run:

```
Set-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Control\FileSystem" -Name
"LongPathsEnabled" -Value 1 -Force
```

4. If you are not a Google employee, you must set the following environment variables to point the depot tools at Visual Studio:

```
DEPOT_TOOLS_WIN_TOOLCHAIN=0
GYP_MSVS_OVERRIDE_PATH="C:\Program Files (x86)\Microsoft Visual
Studio\2019\Community" # (or your location for Visual Studio)
WINDOWSSDKDIR="C:\Program Files (x86)\Windows Kits\10" # (or your location for
Windows Kits)
```

Also, be sure that Python27 is before any other python in your Path.

5. `gclient sync` to update your dependencies.
6. switch to `src/` directory.
7. `python .\flutter\tools\gn --unoptimized` to prepare your build files.
   - If you are only building `gen_snapshot` : `python .\flutter\tools\gn [--unoptimized] --runtime-mode=[debug|profile|release] [--android]` .
8. `ninja -C .\out\<dir created by previous step>` to build.
   - If you used a non-debug configuration, use `ninja -C .\out\<dir created by previous step> gen_snapshot` . Release and profile are not yet supported for the desktop shell.

Compiling for Fuchsia

These steps build the Fuchsia embedding (flutter_runner) and test FAR files that can be deployed to a Fuchsia device.

Note these instructions assume use of `x64` , if `arm64` is needed then just substitute as appropriate.

Note these instructions assume the use of the $FUCHSIA_DIR and $ENGINE_DIR environment variables. These point to the root of your Fuchsia source tree and the root of the Flutter engine source tree (src/ in your Flutter gclient checkout) respectively.

Fuchsia Source Checkout

Testing the Fuchsia embedding requires a Fuchsia source checkout. To get one, go to https://fuchsia.dev/fuchsia-src/get-started and follow the instructions to sync and build a Fuchsia checkout. The `workstation` (e.x. `fx set workstation.nuc`) product uses Flutter as its primary shell and is the primary way of testing Flutter on Fuchsia changes.

IMPORTANT: Dart Version synchronization on Fuchsia

The Fuchsia tree consumes the `flutter_runner` and associated Dart SDK as a set of prebuilts. Flutter apps within the Fuchsia tree are built against the version of the Dart SDK in these prebuilts. Because of this fact, developers must be careful to avoid any skew between the version of Dart VM built into the `flutter_runner` binary and the version of the Dart SDK & VM used by the Flutter toolchain (to compile Flutter apps from Dart code). If there is any mismatch at all between the runner and toolchain, a runtime error results and Flutter won't work at all.

To retrieve the version of Flutter engine in your Fuchsia source tree, run:

```
cat $FUCHSIA_DIR/integration/jiri.lock | grep -A 1 "\"package\":
\"flutter/fuchsia\""
```

Then checkout that git hash in step 2 under "Build the engine".

Build the engine

1. Update the Flutter Engine repo:

```
git -C $ENGINE_DIR/flutter checkout main
git -C $ENGINE_DIR/flutter pull -p upstream
```

2. If you want to checkout a specific git revision:

```
git -C $ENGINE_DIR/flutter checkout <hash>
```

If there are local changes to the Flutter engine that you want to test, make sure to base them on top of this revision.

```
git -C $ENGINE_DIR/flutter checkout -b <desired branch name>
<....make your changes here and commit them...>
```

3. Update your dependencies:

```
cd $ENGINE_DIR
gclient sync
```

4. Prepare your build files:

```
$ENGINE_DIR/flutter/tools/gn --fuchsia --no-lto
```

- `--unoptimized` disables C++ compiler optimizations. On macOS, binaries are emitted unstripped; on Linux, unstripped binaries are emitted
- Add `--fuchsia-cpu=x64` or `--fuchsia-cpu=arm64` to target a particular architecture. The default is x64.
- Add `--runtime-mode=debug` or `--runtime-mode=profile` to switch between JIT and AOT builds. These correspond to a vanilla Fuchsia build and a `--release` Fuchsia build respectively. The default is debug/JIT builds.
- For Googlers, add the `--goma` argument when using goma, and add the `--xcode-symlinks` argument when using goma on macOS.
- Remove `--no-lto` if you care about performance or binary size; unfortunately it results in a *much* slower build.

5. Build a Fuchsia binary:

```
ninja -C $ENGINE_DIR/out/fuchsia_debug_x64
```

- If you used `--unoptimized`, use `ninja -C out/fuchsia_debug_unopt_x64` instead.
- If you used `--runtime-mode=profile`, use `ninja -C out/fuchsia_profile_x64` instead.
- For Googlers, consider also using the `--goma` flag with `gn`, then building with `autoninja` to parallelize the build with Goma.

Deploy to Fuchsia

To test changes, you will first want to make all of the Flutter prebuilts writable:

```
chmod -R +w $FUCHSIA_DIR/prebuilt/third_party/flutter
```

After deploying any wanted changes to the Fuchsia checkout, update your Fuchsia device with any changes you made:

```
cd $FUCHSIA_DIR
fx build && fx ota
```

Deploying flutter_runner

First copy the `flutter_runner` binary itself to your Fuchsia checkout. For standard debug builds:

```
cp $ENGINE_DIR/out/fuchsia_debug_x64/flutter_jit_runner-0.far
$FUCHSIA_DIR/prebuilt/third_party/flutter/x64/debug/jit/flutter_jit_runner-0.far
```

For `--release` Fuchsia builds (you must build Flutter with `--runtime-mode=profile`):

```
cp $ENGINE_DIR/out/fuchsia_profile_x64/flutter_aot_runner-0.far
$FUCHSIA_DIR/prebuilt/third_party/flutter/x64/profile/aot/flutter_aot_runner-0.far
```

If you are changing the native hooks in `dart:ui` , `dart:zircon` , or `dart:fuchsia` you'll also want to update the `flutter_runner_patched_sdk` that is used in your fuchsia checkout (note the use of AOT/release in the destination, that is intentional). Run:

```
cp -ra $ENGINE_DIR/out/fuchsia_debug_x64/flutter_runner_patched_sdk/*
$FUCHSIA_DIR/prebuilt/third_party/flutter/x64/release/aot/flutter_runner_patched_sdk/
```

Deploying debug symbols

Now register debug symbols for all engine artifacts to your Fuchsia checkout:

```
$ENGINE_DIR/fuchsia/sdk/linux/tools/symbol-index add
$ENGINE_DIR/out/fuchsia_debug_x64/.build-id $ENGINE_DIR/out/fuchsia_debug_x64
```

Note: Because of fxbug.dev/45484, `fx log` may have issues symbolize logs on other machines. It is recommended to run `fx log` from the same machine that you build from.

Deploying tests

For any test FAR files, you may publish them to your device using `pm publish` (`flow_tests.far` used as an example; same note as above about the custom `out/` folder applies):

```
$ENGINE_DIR/fuchsia/sdk/linux/tools/pm publish -a -r $FUCHSIA_DIR/$(cat
$FUCHSIA_DIR/.fx-build-dir)/amber-files -f
$ENGINE_DIR/out/fuchsia_debug_x64/flow_tests-0.far
```

```
fx shell run-test-component "fuchsia-
pkg://fuchsia.com/flow_tests#meta/flow_tests.cmx"
```

Make sure to replace both instances of the test name in the "run-test-component" command above with your own.

You can also copy test debug symbols by using the `copy_debug_symbols.py` script and substituting the test binary (such as `flow_unittests`) for the runner binary.

Compiling for the Web

For building the engine for the Web we use the felt tool.

To test Flutter with a local build of the Web engine, add `--local-engine=host_debug_unopt` to your `flutter` command, e.g.:

```
flutter run --local-engine=host_debug_unopt -d chrome
flutter test --local-engine=host_debug_unopt test/path/to/your_test.dart
```

Compiling for the Web on Windows

Compiling the web engine might take a few extra steps on Windows. Use cmd.exe and "run as administrator".

1. Make sure you have Visual Studio installed. Set the following environment variables. For Visual Studio use the path of the version you installed.
   - `GYP_MSVS_OVERRIDE_PATH = "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community"`
   - `GYP_MSVS_VERSION = 2017`
2. Make sure, depot_tools, ninja and python are installed and added to the path. Also set the following environment variable for depot tools:
   - `DEPOT_TOOLS_WIN_TOOLCHAIN = 0`
   - Tip: if you get a python error try to use Python 2 instead of 3
3. `git pull upstream main` in `src/flutter` to update the Flutter Engine repo.
4. `gclient sync` to update your dependencies.
   - Tip: If you get a git authentication errors on this step try Git Bash instead
5. `python .\flutter\tools\gn --unoptimized --full-dart-sdk` to prepare your build files.
6. `ninja -C .\out\<dir created by previous step>` to build.

To test Flutter with a local build of the Web engine, add `--local-engine=host_debug_unopt` to your `flutter` command, e.g.:

```
flutter run --local-engine=host_debug_unopt -d chrome
flutter test --local-engine=host_debug_unopt test/path/to/your_test.dart
```

For testing the engine again use felt tool this time with felt_windows.bat.

```
felt_windows.bat test
```

Troubleshooting Compile Errors

Version Solving Failed

From time to time, as the Dart versions increase, you might see dependency errors such as:

```
The current Dart SDK version is 2.7.0-dev.0.0.flutter-1ef444139c.

Because ui depends on <a pub package> 1.0.0 which requires SDK version >=2.7.0 <3.0.0,
version solving failed.
```

Running `gclient sync` does not update the tags, there are two solutions:

1. under `engine/src/third_party/dart` run `git fetch --tags origin`
2. or run gclient sync with with tags parameter: `gclient sync --with_tags`

See also: [[Debugging the engine]], which includes instructions on running a Flutter app with a local engine.