This page has some hints about debugging the engine.

See also [[Crashes]] for advice on handling engine crashes (specifically around obtaining stack traces, and reporting crashes in AOT Dart code).

Running a Flutter app with a local engine

Once the appropriate version of the engine is built (see [[Compiling the engine]]), run your Flutter app with:

```
$ flutter run --local-engine=XXXX`
```

to run an app with the local engine where `XXXX` should be replaced with the version you wish to use. For example, use `--local-engine=android_debug_unopt` to run a debug android engine or `--local-engine=ios_debug_sim_unopt` to run a debug iOS simulator engine.

It is important to always have a `host_XXXX` version of the engine built when using a local engine since Flutter uses the host build's version of Dart.

Bisecting a roll failure

If the engine roll is failing (see [[Autorollers]]), you can use `git bisect` on the engine repo to track down the offending commit, using the `--local-engine` command as described above to run the failing framework test with each version of the engine.

Tracing OpenGL calls in Skia

All OpenGL calls in Skia are guarded by either the `GR_GL_CALL_NOERRCHECK` or `GR_GL_CALL_RET_NOERRCHECK` macros. Trace events may be added in these macros to trace all GL calls made by Skia, for example in a patch like this.

Due to the number of events traced to the timeline, the trace buffer may be filled up very quickly. Unless you want to see only the traces for the past few frames, use an endless trace buffer (`flutter run --endless-trace-buffer` turns on an endless trace buffer).

Also, make sure to run your application with the `--trace-skia` flag.

Debugging iOS builds with Xcode

Building with `flutter --local-engine` will set a `LOCAL_ENGINE` Xcode build setting in your Flutter application `Generated.xcconfig` file. This will be set until you run `flutter run` again with either a different `--local-engine` option, or with none at all (which will unset it).

You can speed up your workflow by adding the `--config-only` flag to set up the Xcode build settings and plugins, but not compile the app. For example:

```
$ flutter build ios --local-engine ios_debug_unopt --config-only
```

To start debugging, open your Flutter app `ios/Runner.xcworkspace` file in Xcode. Ensure **Product > Scheme > Edit Scheme > Run > Build Configuration** matches your engine runtime mode (defaults to `Debug`).

Scheme > Edit Scheme > Run > Build Configuration" width="900"/>

Add an engine symbol breakpoint via **Debug > Breakpoints > Create Symbolic Breakpoint…**. The **Symbol** field should be the engine symbol you're interested in, like `-[FlutterEngine runWithEntrypoint:]` (note the `-[` prefix has no space).

You can also set a breakpoint directly with lldb by expanding **Flutter > Runner > Supporting Files > main.m** in the Runner Project Navigator. Put a breakpoint in `main()` and start the application by clicking the Run button (CMD + R). Then, set your desired breakpoint in the engine in `lldb` via `breakpoint set -...`.

Debugging Android builds with gdb

See https://github.com/flutter/engine/blob/master/sky/tools/flutter_gdb#L13

Debugging Android builds with Android Studio

First, import the Android embedding into Android studio.

1. Import the `engine/src/flutter/shell/platform/android` subdirectory as a new project. It's important to pick this specific directory. IntelliJ needs this as the root in order to make sense of the package structure.
2. Mark the project as depending on the engine's SDK and Java versions (currently 29 and 8). The option should be visible under `File > Project Structure > Project Settings > Project`.
3. (Optional) Manually tell the IDE to look for any JARs needed by the embedding code in `engine/src/third_party/android_embedding_dependencies/lib` to fix "Missing import" errors. The option should be visible under `File > Project Structure > Modules`, then by selecting the `android` module and clicking on the `Dependencies` tab.

Next, build and run a flutter app using `flutter run --local-engine`. It may be helpful to configure the Android app to wait for the local debugger on start.

Then hit the "Attach debugger" button in Android Studio, click "Show all processes" in the pop up, and select your app from the list and hit OK.

Logging in the engine

Flutter tool will by default parse out any non-error output from the engine. Error logs will be displayed. Logging is handled though the FML library's `logging.h`