This document is intended for developers of third party embedders who wish to package and ship their Flutter applications for AOT mode operation. A third party embedder uses the stable C embedder API to embed Flutter applications on their platform.

Building an AOT Flutter Engine

By default, the Flutter engine packaged for embedders assumes that the mode of operation is JIT. An engine configured for JIT packages a VM that is incompatible with AOT snapshots. Build an AOT engine using the following invocation. This is also the invocation where custom target, sysroot and toolchain selection flags can be specified.

```
./flutter/tools/gn --runtime-mode release <custom target flags as necessary>
```

Note: Throughout this document, the `gn` flags mentioned work fine with other non-runtime mode or configuration selection options (stuff like `--no-lto`,`--unoptimized`, etc..). Such options are particularly useful during development and their use is highly encouraged.

This will produce a Flutter engine configured for AOT mode targeting the host. Note that we are repurposing Flutter's "release" mode policy to prepare an AOT engine. The application of this policy to third party embedders is optional and a decision the authors of such embedders need to make themselves.

Building the Architecture Specific `gen_snapshot`

The binary that converts Dart code to architecture specific AOT instructions is called `gen_snapshot`. A successful invocation of `gen_snapshot` should produce four binary blobs. These are the Dart VM heap and instructions snapshots as well as the isolate heap and instruction snapshots. Refer to the wiki article on engine operation in AOT mode for the purpose of these snapshots.

A specific `gen_snapshot` can only produce AOT instructions for one target architecture. To verify that the `gen_snapshot` you have is producing instructions for the architecture you care about, invoke the same with the `--version` flag. It should produce something like the following.

```
Dart VM version: 2.1.1-dev.2.0.flutter-ac1bf656c4 (Thu Jan 17 16:55:19 2019 +0000) on "macos
```

The final string denotes the host/target pair. In the case of the example above, the host is `macos` and the target `x64`. An example of a `gen_snapshot` that produces instructions for `aarch64` would read "macos_simarm64" (the author is on MacOS X).

It is easiest to just pick the supported Flutter target whose architecture is closest to the one of the embedder. For example, if the target is `armv7`, the following invocation will generate a `gen_snapshot` that is suitably configured for that target.

```
./flutter/tools/gn --android --runtime-mode release
```

Another useful invocation for aarch64 AOT targeted `gen_snapshot` is:

```
./flutter/tools/gn --android --runtime-mode release --android-cpu arm64
```

Important: Calling convention and alignment must be the same (in addition to just the target architecture) for the AOT instructions generated by `gen_snapshot` and the target. Repurposing Flutter's build targets makes sure that all those subtleties are taken care of. But it is still up to the embedder to pick and match the AOT instructions. For example, the `--android` –android-cpu arm64`configured`gen_snapshot`will not generate completely compatible instructions for iOS on`aarch64‘ (even though the target architectures are the same). A mismatch between the snapshot's architecture/ABI and the device's architecture/ABI will be detected when the snapshot is loaded and clearly reported.

Building the AOT Snapshot

There are two separate steps involved in the generation of AOT instructions for the target architecture. First, a target architecture agnostic kernel snapshot (~AST) of the Flutter Dart application needs to be generated. Then, this snapshot is given to `gen_snaphost` which generates the AOT snapshot in the form of the four blobs (~machine code) listed above (and detailed on the wiki page).

The Short and Easy Way

Both iOS and Android AOT modes need to do this step when they build their artifacts. So, if the embedder targets are similar, the workflow supported by the Flutter tooling can be repurposed for custom AOT embedders. For example, in the case of `aarch64` AOT instructions for Android like targets, the following instructions will generated the four AOT blobs in the intermediates.

```
flutter --local-engine <local_engine_configuration> build aot --target-platform android-arm6
```

Note: The `--local-engine` flag is technically optional. However, not specifying the flag will make the tools pick the released version on of the Flutter engine. This version may contain subtle version mismatches with the engine you are using to prepare the `gen_snapshot` binary. So it is safer to just make sure the version are the same.

The result of the invocation will be the generation of the following binary blobs in the `build/aot directory`:

- `vm_snapshot_data`: The VM snapshot data.
- `vm_snapshot_instr`: The VM snapshot instructions.
- `isolate_snapshot_data`: The Isolate snapshot data.
- `isolate_snapshot_instr`: The isolate snapshot instructions.

The Hard Way

To generate the four AOT snapshot blobs directly, you will have to generate the kernel snapshot and then prepare the AOT snapshot manually. The exact flags to use are rather esoteric but self explanatory. And, when necessary, the `-v` flag can be passed to the `flutter build aot` instruction to dump the exact flags used by Flutter. These can then be modified as necessary for the target architecture.

Generating the Kernel Snapshot The following invocation will generate a file called `kernel_snapshot.dill` in the build directory. Make sure you run `flutter packages get` in your project first to fetch all package dependencies.

```
$FLUTTER_ENGINE_OUT_DIR/dart                                    \
    $FLUTTER_ENGINE_OUT_DIR/frontend_server.dart.snapshot       \
    --sdk-root $FLUTTER_ENGINE_OUT_DIR/flutter_patched_sdk/     \
    --strong                                                    \
    --target=flutter                                            \
    --aot                                                       \
    --tfa                                                       \
    -Ddart.vm.product=true                                      \
    --packages .packages                                        \
    --output-dill build/kernel_snapshot.dill                    \
    package:flutter_gallery/main.dart
```

Generating the AOT Snapshot Once the `kernel_snapshot.dill` file has been obtained, `gen_snapshot` can be invoked with the following arguments to generate the four blobs that form the AOT snapshot.

```
$FLUTTER_ENGINE_OUT_DIR/gen_snapshot                                \
    --causal_async_stacks                                           \
    --packages=.packages                                            \
    --deterministic                                                 \
    --snapshot_kind=app-aot-blobs                                   \
    --vm_snapshot_data=build/vm_snapshot_data                       \
    --isolate_snapshot_data=build/isolate_snapshot_data             \
    --vm_snapshot_instructions=build/vm_snapshot_instr              \
    --isolate_snapshot_instructions=build/isolate_snapshot_instr    \
    --no-sim-use-hardfp                                             \
    --no-use-integer-division                                       \
    build/kernel_snapshot.dill
```

Note: The `-no*` flags in the invocation above may not be necessary for all targets and can be skipped. As always, when in doubt, call `flutter build aot -v` after selecting the most similar target and see the flags used by Flutter.

Packaging the AOT Blobs

The four AOT blobs need to be shipped with the application and are necessary for the `FlutterEngineRun` call. Embedders have to make a decision about how best to package and ship these blobs to the target.

On the target at runtime, these blobs need to be mapped into the address space with the following restrictions: * `vm_snapshot_data`: Read-Only. * `vm_snapshot_instr`: Read-Execute. * `isolate_snapshot_data`: Read-Only. * `isolate_snapshot_instr`: Read-Execute.

The responsibility of keeping the mappings alive is upto the embedder. The mappings must be maintained as long as the `FlutterEngine` is running and alive.

Configuring the Engine for AOT Operation

In the `FlutterProjectArgs` struct given the `FlutterEngineRun` call, provide the following options: * `vm_snapshot_data`: Pointer to the read-only VM snapshot mapping. * `vm_snapshot_data_size`: Size of the VM snapshot mapping. * `vm_snapshot_instructions`: Pointer to the read-execute VM instructions mapping. * `vm_snapshot_instructions_size`: Size of the VM instructions mapping. * `isolate_snapshot_data`: Pointer to the read-only isolate snapshot mapping. * `isolate_snapshot_data_size`: Size of the isolate snapshot mapping. * `isolate_snapshot_instructions`: Pointer to the read-execute isolate instructions mapping. * `isolate_snapshot_instructions_size`: Size of the isolate instructions mapping.

Flutter Engine is now running in AOT mode!