// Copyright 2013 The Flutter Authors. All rights reserved.

// Use of this source code is governed by a BSD-style license that can be

// found in the LICENSE file.

#ifndef FLUTTER\_SHELL\_PLATFORM\_WINDOWS\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_FLUTTER\_ENGINE\_H\_

#define FLUTTER\_SHELL\_PLATFORM\_WINDOWS\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_FLUTTER\_ENGINE\_H\_

#include <flutter\_windows.h>

#include <chrono>

#include <memory>

#include <optional>

#include <string>

#include "binary\_messenger.h"

#include "dart\_project.h"

#include "plugin\_registrar.h"

#include "plugin\_registry.h"

namespace flutter {

// An instance of a Flutter engine.

//

// In the future, this will be the API surface used for all interactions with

// the engine, rather than having them duplicated on FlutterViewController.

// For now it is only used in the rare case where you need a headless Flutter

// engine.

class FlutterEngine : public PluginRegistry {

public:

// Creates a new engine for running the given project.

explicit FlutterEngine(const DartProject& project);

virtual ~FlutterEngine();

// Prevent copying.

FlutterEngine(FlutterEngine const&) = delete;

FlutterEngine& operator=(FlutterEngine const&) = delete;

// Starts running the engine at the entrypoint function specified in the

// DartProject used to configure the engine, or main() by default.

bool Run();

// Starts running the engine, with an optional entry point.

//

// If provided, entry\_point must be the name of a top-level function from the

// same Dart library that contains the app's main() function, and must be

// decorated with `@pragma(vm:entry-point)` to ensure the method is not

// tree-shaken by the Dart compiler. If not provided, defaults to main().

bool Run(const char\* entry\_point);

// Terminates the running engine.

void ShutDown();

// Processes any pending events in the Flutter engine, and returns the

// nanosecond delay until the next scheduled event (or max, if none).

//

// This should be called on every run of the application-level runloop, and

// a wait for native events in the runloop should never be longer than the

// last return value from this function.

std::chrono::nanoseconds ProcessMessages();

// Tells the engine that the system font list has changed. Should be called

// by clients when OS-level font changes happen (e.g., WM\_FONTCHANGE in a

// Win32 application).

void ReloadSystemFonts();

// Tells the engine that the platform brightness value has changed. Should be

// called by clients when OS-level theme changes happen (e.g.,

// WM\_DWMCOLORIZATIONCOLORCHANGED in a Win32 application).

void ReloadPlatformBrightness();

// flutter::PluginRegistry:

FlutterDesktopPluginRegistrarRef GetRegistrarForPlugin(

const std::string& plugin\_name) override;

// Returns the messenger to use for creating channels to communicate with the

// Flutter engine.

//

// This pointer will remain valid for the lifetime of this instance.

BinaryMessenger\* messenger() { return messenger\_.get(); }

// Schedule a callback to be called after the next frame is drawn.

//

// This must be called from the platform thread. The callback is executed only

// once on the platform thread.

void SetNextFrameCallback(std::function<void()> callback);

// Called to pass an external window message to the engine for lifecycle

// state updates. Non-Flutter windows must call this method in their WndProc

// in order to be included in the logic for application lifecycle state

// updates. Returns a result if the message should be consumed.

std::optional<LRESULT> ProcessExternalWindowMessage(HWND hwnd,

UINT message,

WPARAM wparam,

LPARAM lparam);

private:

// For access to the engine handle.

friend class FlutterViewController;

// Gives up ownership of |engine\_|, but keeps a weak reference to it.

//

// This is intended to be used by FlutterViewController, since the underlying

// C API for view controllers takes over engine ownership.

FlutterDesktopEngineRef RelinquishEngine();

// Handle for interacting with the C API's engine reference.

FlutterDesktopEngineRef engine\_ = nullptr;

// Messenger for communicating with the engine.

std::unique\_ptr<BinaryMessenger> messenger\_;

// Whether or not this wrapper owns |engine\_|.

bool owns\_engine\_ = true;

// Whether |Run| has been called successfully.

//

// This is used to improve error messages. This can be false while the engine

// is running if the engine was started by creating a view.

bool run\_succeeded\_ = false;

// The callback to execute once the next frame is drawn.

std::function<void()> next\_frame\_callback\_ = nullptr;

};

} // namespace flutter

#endif // FLUTTER\_SHELL\_PLATFORM\_WINDOWS\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_FLUTTER\_ENGINE\_H\_