// 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.

// This file contains the implementations of any class in the wrapper that

// - is not fully inline, and

// - is necessary for all clients of the wrapper (either app or plugin).

// It exists instead of the usual structure of having some\_class\_name.cc files

// so that changes to the set of things that need non-header implementations

// are not breaking changes for the template.

//

// If https://github.com/flutter/flutter/issues/57146 is fixed, this can be

// removed in favor of the normal structure since templates will no longer

// manually include files.

#include <cassert>

#include <iostream>

#include <variant>

#include "binary\_messenger\_impl.h"

#include "include/flutter/engine\_method\_result.h"

#include "include/flutter/method\_channel.h"

#include "include/flutter/standard\_method\_codec.h"

#include "texture\_registrar\_impl.h"

namespace flutter {

// ========== binary\_messenger\_impl.h ==========

namespace {

using FlutterDesktopMessengerScopedLock =

std::unique\_ptr<FlutterDesktopMessenger,

decltype(&FlutterDesktopMessengerUnlock)>;

// Passes |message| to |user\_data|, which must be a BinaryMessageHandler, along

// with a BinaryReply that will send a response on |message|'s response handle.

//

// This serves as an adaptor between the function-pointer-based message callback

// interface provided by the C API and the std::function-based message handler

// interface of BinaryMessenger.

void ForwardToHandler(FlutterDesktopMessengerRef messenger,

const FlutterDesktopMessage\* message,

void\* user\_data) {

auto\* response\_handle = message->response\_handle;

auto messenger\_ptr = std::shared\_ptr<FlutterDesktopMessenger>(

FlutterDesktopMessengerAddRef(messenger),

&FlutterDesktopMessengerRelease);

BinaryReply reply\_handler = [messenger\_ptr, response\_handle](

const uint8\_t\* reply,

size\_t reply\_size) mutable {

// Note: This lambda can be called on any thread.

auto lock = FlutterDesktopMessengerScopedLock(

FlutterDesktopMessengerLock(messenger\_ptr.get()),

&FlutterDesktopMessengerUnlock);

if (!FlutterDesktopMessengerIsAvailable(messenger\_ptr.get())) {

// Drop reply if it comes in after the engine is destroyed.

return;

}

if (!response\_handle) {

std::cerr << "Error: Response can be set only once. Ignoring "

"duplicate response."

<< std::endl;

return;

}

FlutterDesktopMessengerSendResponse(messenger\_ptr.get(), response\_handle,

reply, reply\_size);

// The engine frees the response handle once

// FlutterDesktopSendMessageResponse is called.

response\_handle = nullptr;

};

const BinaryMessageHandler& message\_handler =

\*static\_cast<BinaryMessageHandler\*>(user\_data);

message\_handler(message->message, message->message\_size,

std::move(reply\_handler));

}

} // namespace

BinaryMessengerImpl::BinaryMessengerImpl(

FlutterDesktopMessengerRef core\_messenger)

: messenger\_(core\_messenger) {}

BinaryMessengerImpl::~BinaryMessengerImpl() = default;

void BinaryMessengerImpl::Send(const std::string& channel,

const uint8\_t\* message,

size\_t message\_size,

BinaryReply reply) const {

if (reply == nullptr) {

FlutterDesktopMessengerSend(messenger\_, channel.c\_str(), message,

message\_size);

return;

}

struct Captures {

BinaryReply reply;

};

auto captures = new Captures();

captures->reply = reply;

auto message\_reply = [](const uint8\_t\* data, size\_t data\_size,

void\* user\_data) {

auto captures = reinterpret\_cast<Captures\*>(user\_data);

captures->reply(data, data\_size);

delete captures;

};

bool result = FlutterDesktopMessengerSendWithReply(

messenger\_, channel.c\_str(), message, message\_size, message\_reply,

captures);

if (!result) {

delete captures;

}

}

void BinaryMessengerImpl::SetMessageHandler(const std::string& channel,

BinaryMessageHandler handler) {

if (!handler) {

handlers\_.erase(channel);

FlutterDesktopMessengerSetCallback(messenger\_, channel.c\_str(), nullptr,

nullptr);

return;

}

// Save the handler, to keep it alive.

handlers\_[channel] = std::move(handler);

BinaryMessageHandler\* message\_handler = &handlers\_[channel];

// Set an adaptor callback that will invoke the handler.

FlutterDesktopMessengerSetCallback(messenger\_, channel.c\_str(),

ForwardToHandler, message\_handler);

}

// ========== engine\_method\_result.h ==========

namespace internal {

ReplyManager::ReplyManager(BinaryReply reply\_handler)

: reply\_handler\_(std::move(reply\_handler)) {

assert(reply\_handler\_);

}

ReplyManager::~ReplyManager() {

if (reply\_handler\_) {

// Warn, rather than send a not-implemented response, since the engine may

// no longer be valid at this point.

std::cerr

<< "Warning: Failed to respond to a message. This is a memory leak."

<< std::endl;

}

}

void ReplyManager::SendResponseData(const std::vector<uint8\_t>\* data) {

if (!reply\_handler\_) {

std::cerr

<< "Error: Only one of Success, Error, or NotImplemented can be "

"called,"

<< " and it can be called exactly once. Ignoring duplicate result."

<< std::endl;

return;

}

const uint8\_t\* message = data && !data->empty() ? data->data() : nullptr;

size\_t message\_size = data ? data->size() : 0;

reply\_handler\_(message, message\_size);

reply\_handler\_ = nullptr;

}

} // namespace internal

// ========== method\_channel.h ==========

namespace {

constexpr char kControlChannelName[] = "dev.flutter/channel-buffers";

constexpr char kResizeMethod[] = "resize";

constexpr char kOverflowMethod[] = "overflow";

} // namespace

namespace internal {

void ResizeChannel(BinaryMessenger\* messenger, std::string name, int new\_size) {

auto control\_channel = std::make\_unique<MethodChannel<EncodableValue>>(

messenger, kControlChannelName, &StandardMethodCodec::GetInstance());

// The deserialization logic handles only 32 bits values, see

// https://github.com/flutter/engine/blob/93e8901490e78c7ba7e319cce4470d9c6478c6dc/lib/ui/channel\_buffers.dart#L495.

control\_channel->InvokeMethod(

kResizeMethod, std::make\_unique<EncodableValue>(EncodableList{

EncodableValue(name),

EncodableValue(static\_cast<int32\_t>(new\_size)),

}));

}

void SetChannelWarnsOnOverflow(BinaryMessenger\* messenger,

std::string name,

bool warns) {

auto control\_channel = std::make\_unique<MethodChannel<EncodableValue>>(

messenger, kControlChannelName, &StandardMethodCodec::GetInstance());

control\_channel->InvokeMethod(kOverflowMethod,

std::make\_unique<EncodableValue>(EncodableList{

EncodableValue(name),

EncodableValue(!warns),

}));

}

} // namespace internal

// ========== texture\_registrar\_impl.h ==========

TextureRegistrarImpl::TextureRegistrarImpl(

FlutterDesktopTextureRegistrarRef texture\_registrar\_ref)

: texture\_registrar\_ref\_(texture\_registrar\_ref) {}

TextureRegistrarImpl::~TextureRegistrarImpl() = default;

int64\_t TextureRegistrarImpl::RegisterTexture(TextureVariant\* texture) {

FlutterDesktopTextureInfo info = {};

if (auto pixel\_buffer\_texture = std::get\_if<PixelBufferTexture>(texture)) {

info.type = kFlutterDesktopPixelBufferTexture;

info.pixel\_buffer\_config.user\_data = pixel\_buffer\_texture;

info.pixel\_buffer\_config.callback =

[](size\_t width, size\_t height,

void\* user\_data) -> const FlutterDesktopPixelBuffer\* {

auto texture = static\_cast<PixelBufferTexture\*>(user\_data);

return texture->CopyPixelBuffer(width, height);

};

} else if (auto gpu\_surface\_texture =

std::get\_if<GpuSurfaceTexture>(texture)) {

info.type = kFlutterDesktopGpuSurfaceTexture;

info.gpu\_surface\_config.struct\_size =

sizeof(FlutterDesktopGpuSurfaceTextureConfig);

info.gpu\_surface\_config.type = gpu\_surface\_texture->surface\_type();

info.gpu\_surface\_config.user\_data = gpu\_surface\_texture;

info.gpu\_surface\_config.callback =

[](size\_t width, size\_t height,

void\* user\_data) -> const FlutterDesktopGpuSurfaceDescriptor\* {

auto texture = static\_cast<GpuSurfaceTexture\*>(user\_data);

return texture->ObtainDescriptor(width, height);

};

} else {

std::cerr << "Attempting to register unknown texture variant." << std::endl;

return -1;

}

int64\_t texture\_id = FlutterDesktopTextureRegistrarRegisterExternalTexture(

texture\_registrar\_ref\_, &info);

return texture\_id;

} // namespace flutter

bool TextureRegistrarImpl::MarkTextureFrameAvailable(int64\_t texture\_id) {

return FlutterDesktopTextureRegistrarMarkExternalTextureFrameAvailable(

texture\_registrar\_ref\_, texture\_id);

}

void TextureRegistrarImpl::UnregisterTexture(int64\_t texture\_id,

std::function<void()> callback) {

if (callback == nullptr) {

FlutterDesktopTextureRegistrarUnregisterExternalTexture(

texture\_registrar\_ref\_, texture\_id, nullptr, nullptr);

return;

}

struct Captures {

std::function<void()> callback;

};

auto captures = new Captures();

captures->callback = std::move(callback);

FlutterDesktopTextureRegistrarUnregisterExternalTexture(

texture\_registrar\_ref\_, texture\_id,

[](void\* opaque) {

auto captures = reinterpret\_cast<Captures\*>(opaque);

captures->callback();

delete captures;

},

captures);

}

bool TextureRegistrarImpl::UnregisterTexture(int64\_t texture\_id) {

UnregisterTexture(texture\_id, nullptr);

return true;

}

} // namespace flutter