

Vulkan on Android Developers Guide

Modified: 2015-11-13

This is an early draft of documentation for Android-specific details of using Vulkan. As the API and platform implementation is not yet final, this document will evolve. It is currently based on the same versions as the [LunarG Vulkan SDK 0.9.x](#) release, specifically:

- VK_API_VERSION: [0.170.2](#)
- VK_EXT_KHR_SWAPCHAIN_REVISION: [17](#)
- VK_EXT_KHR_DEVICE_SWAPCHAIN_REVISION: [53](#)
- SPIR-V: [0.99v32](#)

Requirements

Headers and specifications for the API, extensions, and SPIR-V should be obtained from the LunarG SDK or Khronos code repositories. Vulkan-capable Android devices can be obtained from some Khronos-member OEMs and IHVs under NDA.

As with any Android application that uses native code, developers will need the [Android SDK](#) and [NDK](#). The latest versions of the tools are recommended, but no specific platform version is required at this point. This document doesn't describe setting up these tools, [getting started with Android application development](#), or [how to use native code](#) in an Android application.

Using the Vulkan API

Once loaded, the core Vulkan API works the same on Android as on other platforms, and won't be described here.

Loading Vulkan Functions

Vulkan is not yet included in an Android API level, so applications cannot rely on the Vulkan library being present¹ on Android Marshmallow devices. Even when it is included in an API level, applications that want to run on earlier platform versions (e.g. with a fallback to OpenGL ES) will not be able to directly link against the API. In both cases, applications will need to load Vulkan dynamically, and handle the possibility that it might not be present:

¹ Once Vulkan is included in an Android API level, apps that require at least that level will be able to link against the NDK libvulkan.so library and call Vulkan commands without dynamically loading function pointers.

```

void* vulkan_so = dlopen("libvulkan.so", RTLD_NOW | RTLD_LOCAL);
if (!vulkan_so) {
    LOGD("Vulkan not available: %s", dlerror());
    return false;
}

```

Function pointers for the global Vulkan commands (those that do not take a dispatchable object as their first parameter) and `vkGetInstanceProcAddr` must be loaded dynamically:

```

PFN_vkEnumerateInstanceExtensionProperties
vkEnumerateInstanceExtensionProperties =
    reinterpret_cast<PFN_vkEnumerateInstanceExtensionProperties>(
        dlsym(vulkan_so, "vkEnumerateInstanceExtensionProperties"));
PFN_vkEnumerateInstanceLayerProperties vkEnumerateInstanceLayerProperties =
    reinterpret_cast<PFN_vkEnumerateInstanceLayerProperties>(
        dlsym(vulkan_so, "vkEnumerateInstanceLayerProperties"));
PFN_vkCreateInstance vkCreateInstance =
    reinterpret_cast<PFN_vkCreateInstance>(
        dlsym(vulkan_so, "vkCreateInstance"));
PFN_vkGetInstanceProcAddr vkGetInstanceProcAddr =
    reinterpret_cast<PFN_vkGetInstanceProcAddr>(
        dlsym(vulkan_so, "vkGetInstanceProcAddr"));

```

All other Vulkan commands and the commands in the `VK_EXT_KHR_swapchain` and `VK_EXT_KHR_device_swapchain` extensions can be obtained in the same way; function pointers obtained this way can be used with any Vulkan instance.

Alternately, `vkGetDeviceProcAddr` and any commands that take `VkInstance` or `VkPhysicalDevice` as their first parameter can be obtained by calling `vkGetInstanceProcAddr`. The function pointers returned are specific to the instance used to retrieve them, and avoid a dispatch indirection. Similarly, commands that take a `VkDevice`, `VkQueue`, or `VkCommandBuffer` as their first parameter (except `vkGetDeviceProcAddr`) can be obtained from `vkGetDeviceProcAddr`, are specific to a particular device, and avoid a dispatch indirection.

Note: These details have changed in more recent versions of the Vulkan specification. In the initial public release, apps will be able to use `dlsym` to obtain `vkGetInstanceProcAddr`, and then use that to obtain function pointers for all other core and extension commands. `vkGetDeviceProcAddr` will continue to be available and will return device-specific function pointers that avoid dispatch overhead.

Window System Integration

Android uses the `VK_EXT_KHR_swapchain` and `VK_EXT_KHR_device_swapchain` extensions to allow Vulkan to render to on-screen windows. This document doesn't describe how to obtain an `ANativeWindow` representing an Android window; see the NDK [native-activity](#) or [gles3jni](#) samples.

There should be no need to call any `ANativeWindow_*` functions on the `ANativeWindow` directly; all configuration can be done through the Vulkan extensions. As on other platforms, pass a pointer to a `VkSurfaceDescriptionWindowKHR` structure for any `VkSurfaceDescriptionKHR*` parameters. The `VkSurfaceDescriptionWindowKHR` should be filled out as:

```
ANativeWindow* native_window /* = ... */;
const VkSurfaceDescriptionWindowKHR window_description = {
    .sType = VK_STRUCTURE_TYPE_SURFACE_DESCRIPTION_WINDOW_KHR,
    .pNext = nullptr,
    .platform = VK_PLATFORM_ANDROID_KHR,
    .pPlatformHandle = nullptr,
    .pPlatformWindow = native_window,
};
const VkSurfaceDescriptionKHR* surface_description =
    reinterpret_cast<const VkSurfaceDescriptionKHR*>(&window_description);
```

Surface properties and swapchain creation have some platform-specific behaviors. On Android:

- `VkSurfacePropertiesKHR::currentExtent` is the default size of the window; a swapchain with this size will not be scaled during presentation.
- `VkSwapchainCreateInfoKHR::minImageCount` should be set to 3 for best performance on current Android devices when attempting to render at the display refresh rate.
- If `VkSwapchainCreateInfoKHR::imageExtent` is not the same as `VkSurfacePropertiesKHR::currentExtent`, the swapchain images will be scaled to the window size during presentation. The scaling filter is not specified, but is bilinear or better. If the image and surface aspect ratios are different, images will be scaled non-uniformly rather than letterboxed.
- On Android there is no performance advantage to setting `VkSwapchainCreateInfoKHR::clipped` to `VK_TRUE`, though there may be on other platforms.

At the moment, implementation of these extensions is not complete. Only the following is expected to work reliably:

- `VkSwapchainCreateInfoKHR::imageFormat == VK_FORMAT_R8G8B8A8_UNORM`
- `VkSwapchainCreateInfoKHR::preTransform == VK_SURFACE_TRANSFORM_NONE_KHR`
- `VkSwapchainCreateInfoKHR::presentMode == VK_PRESENT_MODE_FIFO_KHR`
- `VkSwapchainCreateInfoKHR::oldSwapchain == VK_NULL_HANDLE`

Support for additional image formats, pre-transformed images, mailbox presentation mode (but not immediate presentation), and swapchain re-creation are all expected to work before the initial public release.

Validation Layers

The LunarG validation layers and `DEBUG_REPORT` extension are usable on Android.

Obtaining Layers

Layers can be downloaded from the Khronos [GitLab LoaderAndTools repository](#), using branch 'sdk-0.9'. To build the layers for all ABIs:

```
LoaderAndTools$ cd buildAndroid
buildAndroid$ ./android-generate.sh
buildAndroid$ ndk-build
```

Layer libraries will be in `libs/<abi>`.

Installing Layers

Layer libraries included in the APK native library directory will be enumerated and can be enabled even in non-debuggable apps. Copy them into the appropriate `libs/<abi>` directory (e.g. via a build rule) and they will automatically be included in the APK by 'ant', and extracted during installation.

TODO: Figure out how this works with the Gradle build system used by Android Studio.

For [debuggable apps](#), layer libraries in `/data/local/tmp/vulkan/` will also be enumerated and can be enabled.

TODO: Confirm this works as expected on user device builds. We may need to change the path (e.g. something under `/sdcard/`), and the application probably has to have `<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE">`.

Most developers will want to explicitly enable/disable validation layers programmatically, and provide their own message handler callback via the `DEBUG_REPORT` extension. However, if the app is debuggable, validation layers can be enabled and `ERROR` and `WARN` messages sent to logcat without modifying the app:

```
$ adb shell setprop debug.vulkan.layer.<n> <layer name>
$ adb shell setprop debug.vulkan.enable_callback 1
# For example:
# adb shell setprop debug.vulkan.layer.0 ParamChecker
# adb shell setprop debug.vulkan.layer.1 DrawState
```

When these properties are defined, the loader will add the named layers to the enabled layer list, ordered by increasing `<n>`.