Vulkan SDK Version Compatibility

(Originally Published May 2018, Latest update April 2020)

Vulkan was originally released in March of 2016. Since that time the Khronos Vulkan Working Group continues to update the Vulkan API. When new Vulkan releases become available, the version number associated with the API gets incremented. This document was originally published in May of 2018 to share the impact of the new Vulkan 1.1 release on applications written with Vulkan 1.0. We've updated this white paper to explain how new Vulkan 1.x releases may impact applications.

Major and Minor Releases

Vulkan has three components to its release numbering. The first number indicates the Major version, the second number is the Minor version, and the third number is the Patch version.

Vulkan 1.2.131

- The Major version here is 1
- The Minor version here is 2
- The Patch version here is 131

The Major version is used to indicate a major change to the API that may or may not be compatible with other Major versions of the API. For example, a library or tool written for Vulkan 5.0 (hypothetical future release), would not necessarily be compatible with any earlier version of Vulkan. Therefore, whenever a new Major version of the Vulkan API is released, you will require newer versions of the entire tool chain to work with the newer API.

On the other hand, the Minor version indicates a backwards compatible change to the API. For example, Vulkan 1.0 and Vulkan 1.1 libraries and layers may be used by Vulkan 1.0 applications. Likewise, a Vulkan 1.2 loader could be used by Vulkan 1.0, 1.1 and 1.2 applications. Each new Minor release simply adds additional functionality to the API that may be used without modifying pre-existing applications. Of course, since new functionality is added, a Vulkan 1.2 application requires a Vulkan 1.2 loader and layers to properly function.

The Patch version is simply used to indicate what specific weekly release of the Vulkan specification and headers was used when generating the files. The Patch version should be ignored in general application development.

Does the latest Vulkan version 1 SDK support my older Vulkan 1.x application?

Since Minor releases are backwards compatible, there is no need for SDKs to be released for older Minor release versions of the API. Minor releases such as Vulkan 1.1 and 1.2 add additional functionality to the API without modifying the behavior of anything already using Vulkan 1.0 functionality. Therefore, it guarantees compatibility for applications written for earlier 1.x versions of Vulkan:

1. Application binaries built from a 1.0 Vulkan header will work with the latest Vulkan 1.x versions of the validation layers, Vulkan runtime (loader), and drivers.
2. Application source written to use an earlier Minor release Vulkan headers (such as those written for Vulkan 1.0 or 1.1) can be built with the latest release of the Vulkan 1.x SDK.

LunarG no longer plans to release Vulkan SDKs based on older Minor releases of the Vulkan headers because of this compatibility. To continue to get the latest validation layers and other SDK components, you can use the latest Vulkan 1.x versioned SDKs and be confident that the validation layers, tools, and Vulkan runtime will properly work with your applications written against an earlier Minor release of Vulkan.

As well, Windows driver updates from independent hardware vendors (IHVs) will continue to bundle the Vulkan Runtime and they will be moving to the latest Minor release of Vulkan 1.x as soon as possible.

How to migrate your applications to the latest Vulkan 1.x Minor release

Starting with Vulkan 1.1, the *VkApplicationInfo* substructure of *VkInstanceCreateInfo* is no longer optional if you want to create a Vulkan 1.1 (or newer) application. There is now a process for properly creating a Vulkan application for Vulkan 1.1 and newer:

1. First, find out what API version the runtime supports for Instances:
    a. To do this, you must determine if the **vkEnumerateInstanceVersion** entry-point is available on your system.

```
PFN_vkEnumerateInstanceVersion
      pEnumInstanceVersion =
            reinterpret_cast<PFN_vkEnumerateInstanceVersion*>(
                  vkGetInstanceProcAddr(VK_NULL_HANDLE,
                        "vkEnumerateInstanceVersion"));
```

b. If **pEnumInstanceVersion** is NULL, then your system can only support Vulkan 1.0.

c. If **pEnumInstanceVersion** is not NULL, then you must query the API version that your Vulkan loader can support for *VkInstance* objects by calling the function.

```
uint32 api_version = VK_MAKE_VERSION(1, 0, 0);
if (NULL != pEnumInstanceVersion) {
      // Call down and get the actual Vulkan API Version
      // supported.
      // NOTE: It should replace what we originally put in.
      pEnumInstanceVersion(&api_version);
}
```

This is the maximum API version an instance can be created for on a system. If the returned version is greater-than or equal-to Vulkan 1.2 you may create a Vulkan 1.2 Instance. If not, you may only create an application using the largest minor version returned by the **vkEnumerateInstanceVersion** function pointer.

For example:

```
if (VK_MAKE_VERSION(1, 2, 0) <= apiVersion) {
      // 1.2 or newer is available
} else if (VK_MAKE_VERSION(1, 1, 0) <= apiVersion) {
      // 1.1 or newer is available
} else {
      // Only Vulkan 1.0 is available
}
```

2. Next, if Vulkan 1.2 instances are supported, you must create a *VkApplicationInfo* structure, and set the “apiVersion” field to the following:

```
VkApplicationInfo myApplicationInfo = {};
...
myApplicationInfo.apiVersion = VK_MAKE_VERSION(1, 2, 0);
```

3. Then, you must set the *VkInstanceCreateInfo* **pApplicationInfo** to point to the above application info struct:

```
VkInstanceCreateInfo myInstanceCreateInfo = {};
…
myInstanceCreateInfo.pApplicationInfo = &myApplicationInfo;
```

4. Finally, call **vkCreateInstance** as you normally would with your *VkInstanceCreateInfo* structure.

Ensuring that your physical devices support the latest Minor release of Vulkan

An additional process is necessary to ensure that your physical devices support the latest Minor release of Vulkan:

1. Once you have created a Vulkan instance at the proper release version, you still must check which physical devices support Vulkan at that specific Minor release. This check is necessary because multiple Vulkan capable devices may be available on your system. While one physical device may have been updated to support the latest Minor release, one or more of the others may not yet support the latest Minor release. To verify the version on any physical device, call:

```
VkPhysicalDeviceProperties properties = {};
…
vkGetPhysicalDeviceProperties(physicalDevice, &properties);
```

2. The supported API version of this physical device will be given in the “apiVersion” field of the *VkPhysicalDeviceProperties* structure and will take the same format as the “apiVersion” field in **vkEnumerateInstanceVersion**.

Code Snippet

[Based on the **“vulkan_1_1_flexible.cpp”** sample in the Vulkan SDK]

```
VkInstance instance = VK_NULL_HANDLE;

// Keep track of the major/minor version we can actually use
uint16_t using_major_version = 1;
uint16_t using_minor_version = 0;
std::string using_version_string = "";

// Set the desired version we want
uint16_t desired_major_version = 1;
uint16_t desired_minor_version = 1; // Set this to the version you desire
uint32_t desired_version = VK_MAKE_VERSION(
      desired_major_version,
```

```
      desired_minor_version,
      0);

std::string desired_version_string = "";
desired_version_string += std::to_string(desired_major_version);
desired_version_string += ".";
desired_version_string += std::to_string(desired_minor_version);
VkInstance instance = VK_NULL_HANDLE;
std::vector<VkPhysicalDevice> physical_devices_desired;

// Determine if the new instance version command is available
PFN_vkEnumerateInstanceVersion pEnumerateInstanceVersion =
      (PFN_vkEnumerateInstanceVersion)vkGetInstanceProcAddr(
            VK_NULL_HANDLE,
            "vkEnumerateInstanceVersion");

// If the command exists, query what version the Vulkan instance supports
uint32_t api_version = 0;
uint16_t instance_major_version = 1;
uint16_t instance_minor_version = 0;
if (NULL != pEnumerateInstanceVersion &&
    VK_SUCCESS == pEnumerateInstanceVersion(&api_version)) {

    // Translate the version into major/minor for easier comparison
    instance_major_version = VK_VERSION_MAJOR(api_version);
    instance_minor_version = VK_VERSION_MINOR(api_version);
    std::cout <<
            "Instance support detected for Vulkan " <<
            instance_major_version << "." << instance_minor_version << "\n";
}

// Check current version against what we want to run
if (instance_major_version > desired_major_version ||
    (instance_major_version == desired_major_version &&
     instance_minor_version >= desired_minor_version)) {

    // Initialize the VkApplicationInfo structure with the version
    // of the API we're intending to use
    VkApplicationInfo app_info = {};
    app_info.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    app_info.pNext = NULL;
    app_info.pApplicationName = APP_SHORT_NAME;
    app_info.applicationVersion = 1;
    app_info.pEngineName = APP_SHORT_NAME;
    app_info.engineVersion = 1;
    app_info.apiVersion = desired_version;
```

```
    // Initialize the VkInstanceCreateInfo structure
    VkInstanceCreateInfo inst_info = {};
    inst_info.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    inst_info.pNext = NULL;
    inst_info.flags = 0;
    inst_info.pApplicationInfo = &app_info;
    inst_info.enabledExtensionCount = 0;
    inst_info.ppEnabledExtensionNames = NULL;
    inst_info.enabledLayerCount = 0;
    inst_info.ppEnabledLayerNames = NULL;

    // Attempt to create the instance
    if (VK_SUCCESS != vkCreateInstance(&inst_info, NULL, &instance)) {

        std::cout << "Unknown error creating " <<
                  desired_version_string << " Instance\n";
        exit(-1);
    }

    // Get the list of physical devices
    uint32_t phys_dev_count = 1;
    if (VK_SUCCESS != vkEnumeratePhysicalDevices(instance,
                              &phys_dev_count, NULL) ||
        phys_dev_count == 0) {

        std::cout << "Failed searching for Vulkan physical devices\n";
            exit(-1);
    }
    std::vector<VkPhysicalDevice> physical_devices;
    physical_devices.resize(phys_dev_count);
    if (VK_SUCCESS != vkEnumeratePhysicalDevices(instance,
                              &phys_dev_count, physical_devices.data()) ||
        phys_dev_count == 0) {

        std::cout << "Failed enumerating Vulkan physical devices\n";
        exit(-1);
    }

    // Go through the list of physical devices and select only
    // those that are capable of running the API version we want.
    for (uint32_t dev = 0; dev < physical_devices.size(); ++dev) {

        VkPhysicalDeviceProperties physical_device_props = {};
        vkGetPhysicalDeviceProperties(physical_devices[dev],
                        &physical_device_props);
        if (physical_device_props.apiVersion >= desired_version) {
            physical_devices_desired.push_back(physical_devices[dev]);
```

```
        }
    }

    // If we have something in the desired version physical device
    // list, we're good
    if (physical_devices_desired.size() > 0) {
        using_major_version = desired_major_version;
        using_minor_version = desired_minor_version;
    }
}

if (using_major_version < desired_major_version ||
    (using_major_version == desired_major_version &&
     using_minor_version < desired_minor_version)) {

    using_version_string += std::to_string(using_major_version);
    using_version_string += ".";
    using_version_string += std::to_string(using_minor_version);

    std::cout <<
        "Determined this system can only use Vulkan API version " <<
        using_version_string <<
        " instead of desired version " << desired_version_string <<
        std::endl;

    vkDestroyInstance(instance, nullptr);
    exit(-1);
}

std::cout <<
      "Determined that this system can run desired Vulkan API version " <<
      desired_version_string << std::endl;

// You can now use Vulkan desired_major_version.desired_minor_version!
```

Supporting 1.0 and later Vulkan Loaders

If you intend to support Vulkan 1.0 loaders as well as any newer Vulkan loaders, you can't link directly to any Vulkan 1.1, 1.2, or newer commands. Otherwise, when your Vulkan application attempts to utilize the newer commands on a user's system, it will crash since the dynamic linking will fail. Because of this, if you do intend to support users who may have older versions of the Vulkan loader installed, you must build your own dispatch table of newer non-Vulkan 1.0

commands using **vkGetInstanceProcAddr** and **vkGetDeviceProcAddr** and then validate that the commands are present as well as verifying the Vulkan version as mentioned above.

Since there is a likelihood of your application reaching a user who still has a Vulkan 1.0 loader on their system, it is best to make sure you consider this during your creation process.

In Summary

You cannot legally use any Vulkan functionality beyond Vulkan 1.0 if you do any of the following:

1. Fail to successfully call **vkEnumerateInstanceVersion**.
2. Set **myInstanceCreateInfo.pApplicationInfo = NULL** when creating your instance.
3. Set **myApplicationInfo.apiVersion** incorrectly when creating your instance. Incorrect values include:
    a. **0**
    b. **VK_MAKE_VERSION(1, 0, <any_integer>)**

Remember, to legally use any Vulkan 1.x functionality beyond Vulkan 1.0 in your application you **should**:

1. Query the **vkEnumerateInstanceVersion** function pointer by calling **vkGetInstanceProcAddr**
2. Call returned **vkEnumerateInstanceVersion** function pointer
    a. Verify that it returns **VK_SUCCESS**
    b. Determine that the API version returned is greater than or equal to **VK_MAKE_VERSION(1, <*target_minor_version*>, 0)**
3. Create an instance with **VkApplicationInfo** defined and the “apiVersion” field set to **VK_MAKE_VERSION(1, <*target_minor_version*>, 0)**
4. Select a physical device that supports Vulkan 1.***<target_minor_version>*** by querying device support through **vkGetPhysicalDeviceProperties** and checking the “apiVersion” field of the filled in *VkPhysicalDeviceProperties* structure.

All these steps are required to properly initialize the latest Vulkan 1.x. If you don’t perform all of these steps, you can only be guaranteed that Vulkan 1.0 functionality is present.