The Vulkan Profiles Toolset Solution

Creating portable Vulkan Applications in terms of Vulkan capabilities

Christophe Riccio, LunarG

February 2022

Why do Vulkan Profiles matter?

From the perspective of LunarG, the Vulkan Profiles aren't aiming at building a perfectly homogenous Vulkan ecosystem against an overly heterogeneous ecosystem: Vulkan remains a cross-platform industry standard enabling developers to target a wide range of devices with the same graphics API.

Instead, *Vulkan Profiles* look at the problem with a different paradigm: a Vulkan application is not designed to target the entire ecosystem, it has a domain of relevance.

Explicit Vulkan capabilities requirements

The *Vulkan Profiles* make the Vulkan capabilities requirements explicit between the application and the domain and we should expect a co-evolution formalized by future profiles versions.

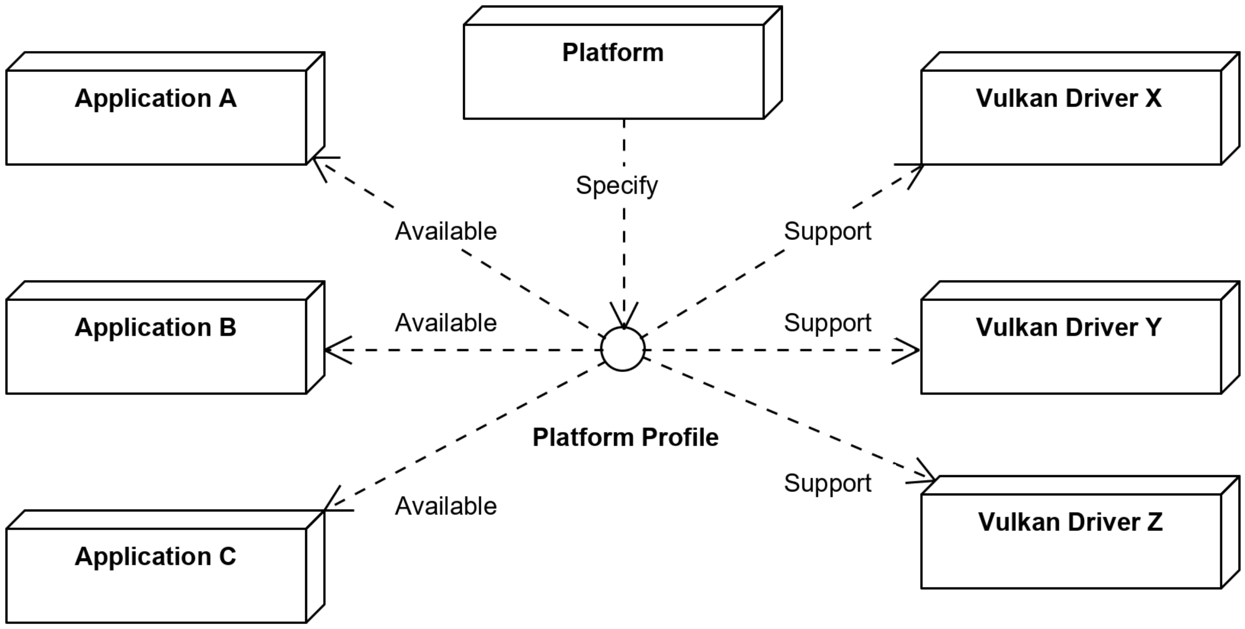
Vulkan Profiles are a tool to help create a portable Vulkan application within a specific Vulkan domain.

Conceptually, *Vulkan Profiles* can be understood as the explicit expression and formalization of Vulkan capabilities requirements and can provide clear communication of these requirements within the Vulkan community.

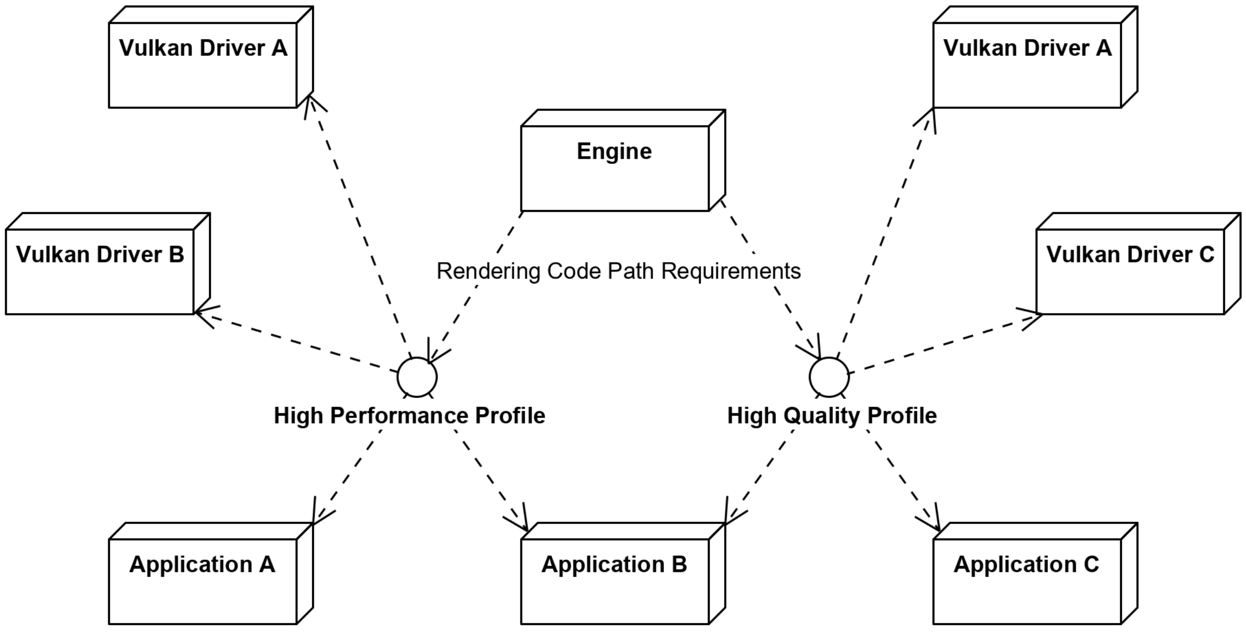
Examples of Vulkan Profiles use cases

Vulkan Profiles can be applied for a multitude of use cases, including:

- Roadmap profiles: to express guidance on the future direction of Vulkan devices.
- Platform profiles: to express the Vulkan support available on different platforms.
- Device profiles: to express the Vulkan support of a single Vulkan driver for a Vulkan device.
- Architecture profiles: to express the Vulkan support of a class of GPUs.
- Engine profiles: to express some rendering code paths requirements of an engine.
- Etc.



For example, a Platform Profile may be used by a platform vendor to specify to the Vulkan application developers the minimum level of Vulkan capabilities support available on its platform.



For example, an engine can specify the requirements of rendering code paths hence the requirements that leveraging such code paths imply for Vulkan applications.

Vulkan Profiles from Vulkan developers' point of view

LunarG provides the *Vulkan Profiles Toolset* as part of the *Vulkan SDK* so that Vulkan application developers may leverage *Vulkan Profiles* during Vulkan application development and delivery. Developers can create portable Vulkan applications in terms of Vulkan capabilities which include extensions, features, properties, formats, and queues requirements.

The *Vulkan Profiles Toolset* is effectively a collection of components for Vulkan application developers to build portable Vulkan applications using *Vulkan Profiles*. Of course, the Vulkan Profiles only handle portability in terms of Vulkan capabilities.

The *Vulkan Profiles Toolset* is still under development at the **BETA** stage.

The Vulkan Profiles Toolset Components

The Vulkan Profiles Toolset includes the following components:

- The Vulkan Profiles schema
    - A JSON data format to exchange Vulkan capabilities: extensions, features, properties, formats, and queue properties.
    - Each revision of Vulkan API is represented by a schema that supersedes older versions of Vulkan API.
    - The schema covers Vulkan 1.3 and all extensions.
- The Vulkan Profiles layer
    - A layer used during application development to ensure adherence to the requirements of a chosen Vulkan Profile.
    - It simulates but doesn't emulate Vulkan capabilities. It works together with the Validation layer which reports errors when using capabilities not exposed by the Vulkan developer system.
    - The layer requires a Vulkan 1.0 driver that supports the `VK_KHR_get_physical_device_properties2` extension.
- The Vulkan Profiles library
    - A header-only, C++ library to use Vulkan Profiles in Vulkan applications.
    - The library allows checking Profiles support on a device and creating a `VkDevice` instance with the profile features and extensions enabled.
    - The library requires a Vulkan 1.0 driver that supports the `VK_KHR_get_physical_device_properties2` extension.
    - A Vulkan sample is available for demonstrating Vulkan Profiles library usage.
- The Vulkan Profiles comparison table
    - Human-readable format of Vulkan Profiles in a table to enable comparison.

Furthermore, the Vulkan SDK includes implementations of some Vulkan Profiles using the Vulkan Profiles JSON Schema:

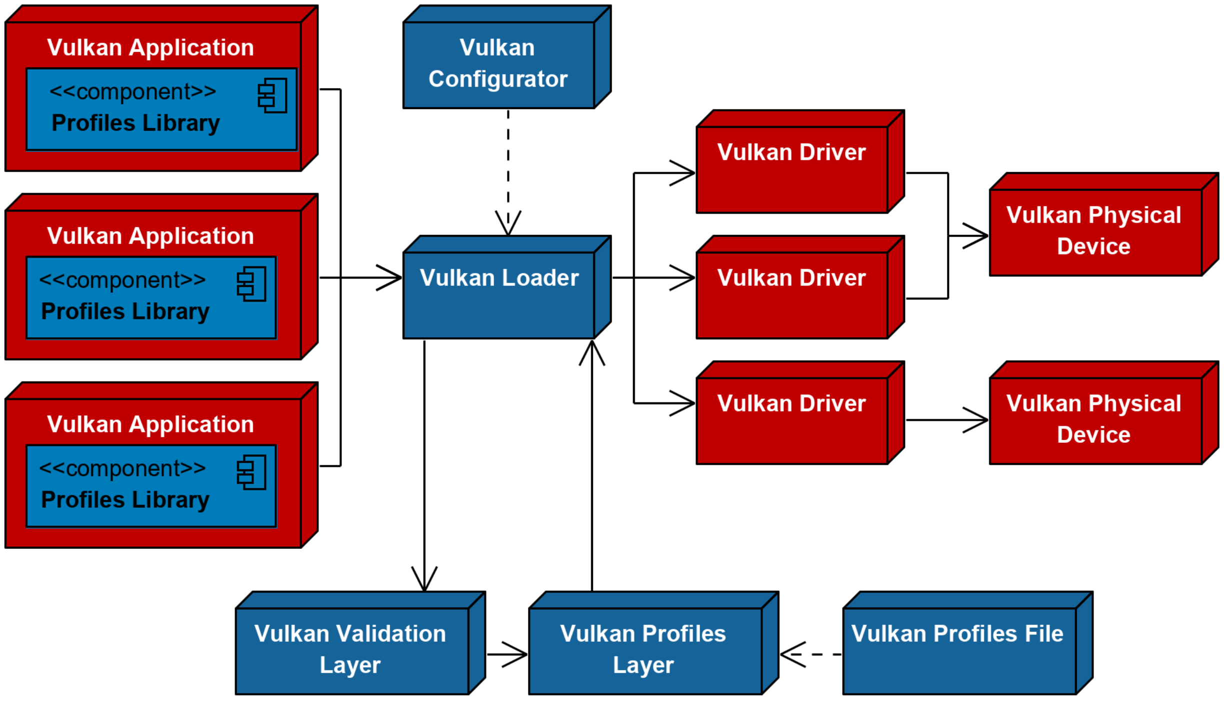
- `VP_KHR_roadmap_2022.json`
- `VP_ANDROID_baseline_2021.json`
- `VP_LUNARG_desktop_portability_2021.json`

Each profiles JSON file may contain multiple variants of a profile. For example `VP_LUNARG_desktop_portability_2021.json` specifies `VP_LUNARG_desktop_portability_2021` profile for Windows and Linux but also `VP_LUNARG_desktop_portability_2021_subset` profile for macOS, which has specific additional requirements regarding `VK_KHR_portability_subset` extension.

The *Vulkan Profiles API* is not part of the Vulkan specification but part of a library for a very specific reason: we wanted the Vulkan Profiles solution to be effectively usable on day one. Hence, it should work with current Vulkan devices currently owned by users.

The solution to this problem is to not deliver the Vulkan Profiles API through the Vulkan drivers that can't be updated easily on many platforms, but to deliver the Vulkan Profiles API as part of the Vulkan application codebase.

As a result, the Vulkan Profiles library is compatible with any Vulkan 1.0 drivers that support `VK_KHR_get_physical_device_properties2` and can be leveraged simply by including it in the Vulkan application codebase.



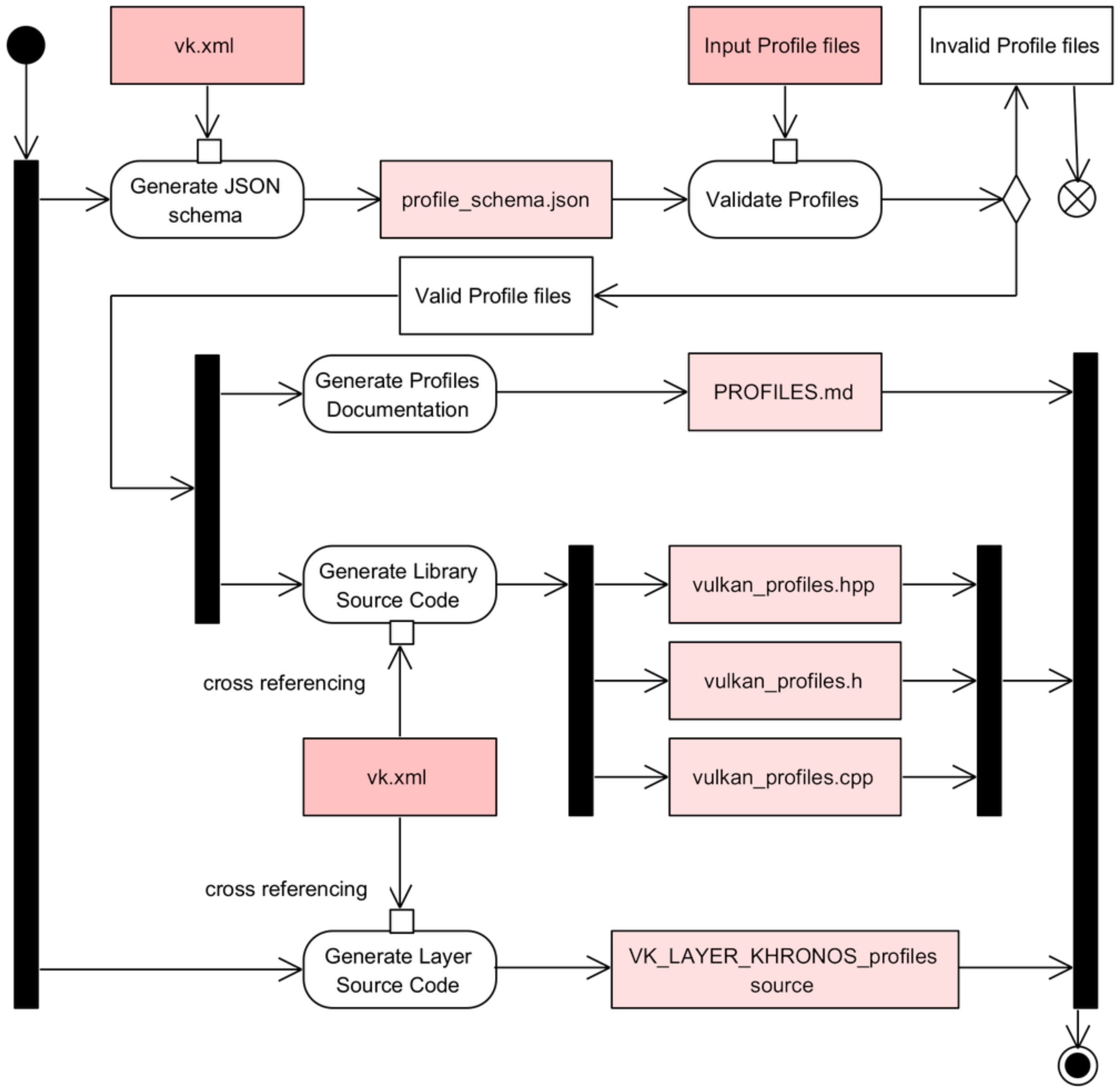
The Vulkan Profiles Toolset components on the Vulkan developer system

The Vulkan Profiles Toolset Code Generation

Considering the complexity of the Vulkan ecosystem, there isn't a single Vulkan Profile that can fit all needs. As a result, on top of the predefined Vulkan Profiles, the Vulkan Profiles Toolset solution is designed around the idea of code generation.

The Vulkan Profiles Toolset is also generated against vk.xml (the canonical representation of the Vulkan specification) and Vulkan Profiles. This design guarantees that any Vulkan developer can regenerate the entire Vulkan Profiles Toolset solution with any new Vulkan Header update or any set of Vulkan Profiles JSON files.

The diagram on the following page shows the Vulkan Profiles Toolset generation pipeline with every produced component:



Using the *Vulkan Profiles schema*

What is the *Vulkan Profiles schema?*

The *Vulkan Profiles schema* is a JSON file generated using the vk.xml file. For each *Vulkan Header* revision, we can generate a *Vulkan Profiles schema*. Generating a new *Vulkan Profiles schema* may be necessary to leverage Vulkan capabilities introduced with a new *Vulkan Header* revision.

The role of the *Vulkan Profiles schema* is to ensure that profile files, created in the ecosystem, will be syntactically valid files. Specifically, the schema validated the name of the structures and structures members so there can't be any spelling errors. Similarly, the schema lists all valid enum values.

Validating *Vulkan Profiles* JSON files against the schema can be performed using any tools typically used for this purpose, including the web-based validators such as:

- http://www.jsonschemavalidator.net/
- https://json-schema-validator.herokuapp.com/
- https://jsonschemalint.com/#/version/draft-04/markup/json/

But also C++ libraries such as Valijson and most likely any JSON schema validation library already integrated into your engine codebase.

Finding Vulkan Profiles files

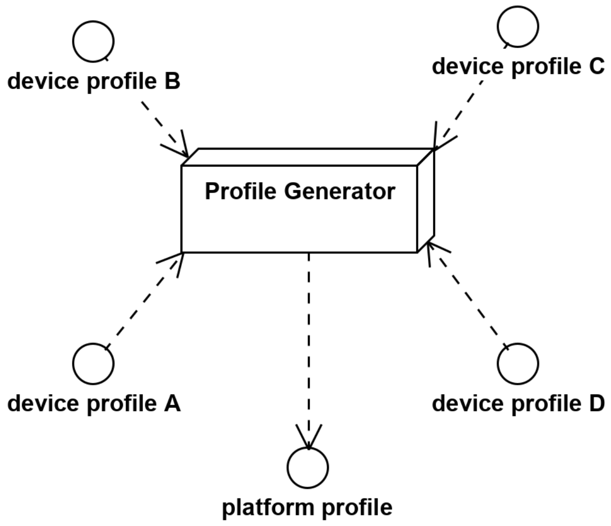
As described previously, there can be many different kinds of Vulkan Profiles files.

The Vulkan SDK and Vulkan Profiles repository include available *Roadmap and Platform profiles* that will help Vulkan developers to target specific Vulkan ecosystem domains. LunarG is planning to continue populating the Vulkan SDK and Vulkan Profiles repository with useful profiles.

GPUInfo.org provides *Device Profiles* thanks to an exporter following the *Vulkan Profiles schema* for any Vulkan driver report. These device profiles can be useful to check that a Vulkan application is working correctly with the capabilities of older Vulkan drivers. Additionally, collecting multiple *Devices Profiles* could be done to help create a *Platform profile* or an *Engine profile* but creating such profiles remains a manual process.

In the next major SDK release, we are planning to improve Vulkaninfo to provide an export of the system device Vulkan capabilities following the *Vulkan Profiles schema*.

Furthermore, we want to provide a profile generator tool that will be capable of merging multiple profiles together but this remains a manual process that can be facilitated by regenerated Vulkan SDK documentation.



Concept to generate a profile from device profiles but this remains a manual process.

The *Vulkan Profiles* documentation

The *Vulkan Profiles* documentation can be used to easily read the requirements of a profile and compare multiple profiles side by side.

It can easily be regenerated and augmented with more profiles by simply copying the list of *Vulkan Profiles* files we want to document into the Profiles directory in the source.

Limitations of *Vulkan Profiles schema* validation

Unfortunately, a profile file would pass schema validation even if it requires a minimum Vulkan API version but uses the Vulkan structure that was introduced after that specified *Vulkan Header* revision. This makes the profile definition infringing its own requirements which is indeed incorrect. Unfortunately, vk.xml is a snapshot of a *Vulkan Header* revision for a specific Vulkan API version so it doesn't store with which *Vulkan Header* revision a Vulkan capability was introduced.

For this reason, we recommend using the *Vulkan Profiles schema* revision that matches the *Vulkan Profile* API version minimum requirements. Many revisions of the profiles schema for *Vulkans Header* are available on Khronos.org.

Similarly, the Profiles JSON Schema can't validate a lot of the semantic aspects. For example, the `runtimeDescriptorArray` Vulkan feature can be enabled using multiple structures: `VkPhysicalDeviceDescriptorIndexingFeaturesEXT`, `VkPhysicalDeviceDescriptorIndexingFeatures`, and `VkPhysicalDeviceVulkan12Features`. Syntactically, all these structures can be used simultaneously for the definition of a profile but what happens if they are specified with different values?

To address these cases, we can use the *Vulkan Profiles* layer which reports warning messages.

Using the *Vulkan Profiles layer*

Simulation vs. Emulation

The primary function of the *Vulkan Profiles Layer* is simply to simulate the capabilities of a device, modifying device responses to Vulkan query function calls by the application. Of course, the underlying device or driver function is never actually changed, they merely appear to have the capabilities of a different device or driver.

This is different from emulation which would change the actual behavior of the underlying device or driver to match that of a different device or driver. In all but one case, the *Profiles layer* simulates changes and leaves it up to the *Validation Layer* to inform the developer about functions that do not adhere to the proper limits.

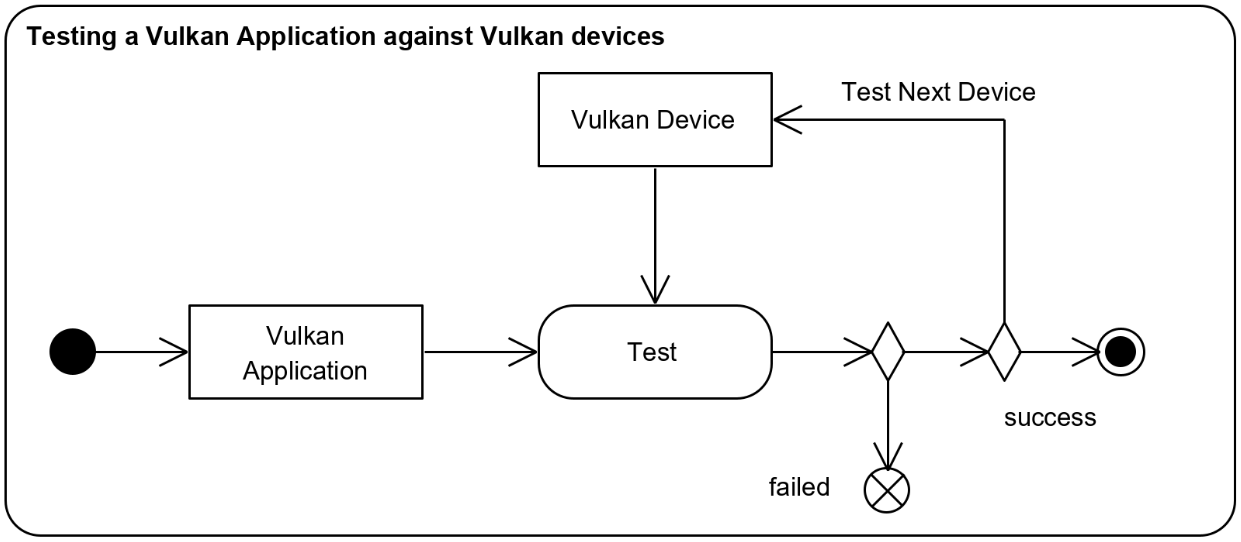
The one exception is portability subset extension emulation, which causes the *Profiles Layer* to add the `VK_KHR_portability_subset` extension to the device extensions list, and pre-populate the `VkPhysicalDevicePortabilitySubsetPropertiesKHR` and `VkPhysicalDevicePortabilitySubsetFeaturesKHR` structures of this extension with default values.

Effectively, the *Vulkan Profiles layer* is used for downgrading the Vulkan application developers' system capabilities.

Reducing Vulkan application development time

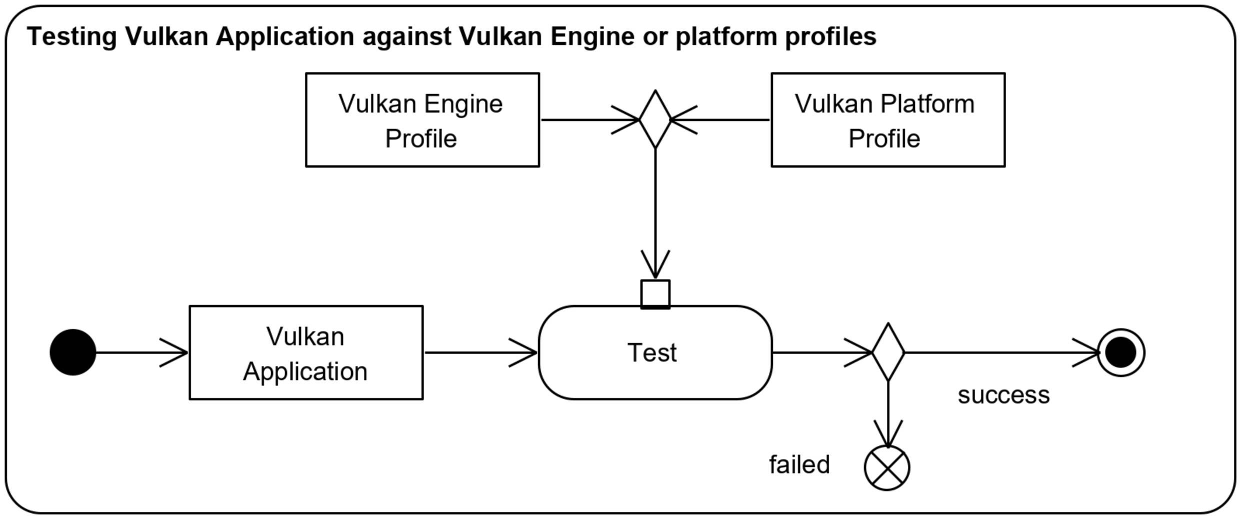
The Vulkan Profiles layer is expected to be used during Vulkan application development and testing. It aims at drastically improving the way we test our Vulkan applications across a wide range of hardware capabilities.

Typically when developing a Vulkan application, we need to check that our Vulkan application works on a set of platforms, devices, and even Vulkan driver versions. However, this process can be particularly tedious and time-consuming which translates into cutting some testing or reducing support of old Vulkan drivers.



Typical testing strategy, a device and driver at a time

The Vulkan Profiles and the Vulkan Profiles layer enable a new strategy: instead of checking capabilities of a device and driver set at a time, the solution allows checking an entire range of devices and drivers at a time by checking against a Vulkan profile that represents all these devices or better, the Vulkan application requirements directly.



Testing Vulkan capabilities support against a Vulkan profile

The *Vulkan Profiles* layer can be used for many use cases:

- Using C.I. to ensure that the Vulkan application never adds unintentional Vulkan capabilities requirements.
- Verifying that the Vulkan application falls back correctly when a driver doesn't support a capability without updating the drivers or recompiling the Vulkan application.

- Verifying whether a Vulkan application behavior on a machine is due to the capabilities of that machine.
- Verifying the Vulkan application works on a less capable Vulkan device than the Vulkan developer device.
- Verifying the Vulkan Profile is well formed, with no unexpected duplicated references of Vulkan capabilities.
- Excluding device extensions and image formats to validate the robustness of the Vulkan application.
- Etc.

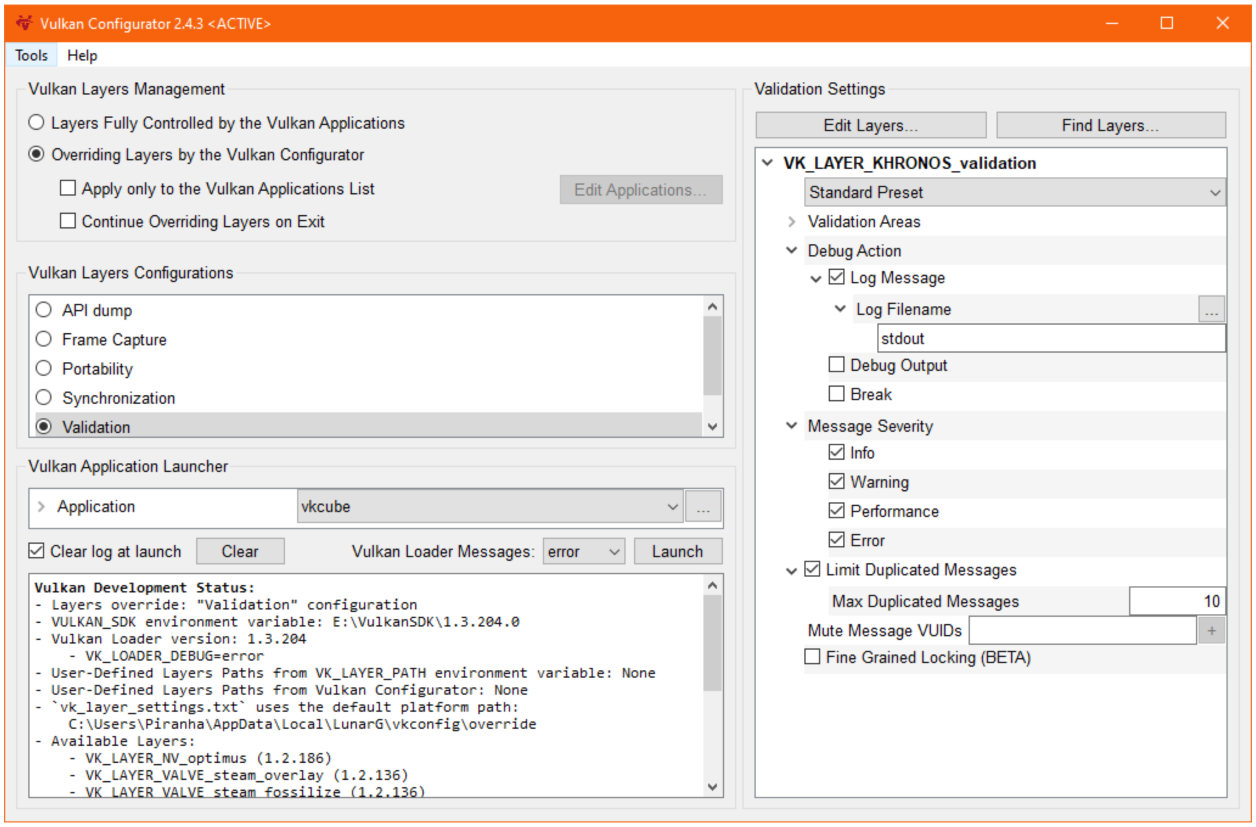
Enabling the *Profiles layer* using *Vulkan Configurator*

We highly recommend using *Vulkan Configurator* to use *Vulkan Layers* to improve Vulkan application development effectiveness.

Before *Vulkan Configurator*, a Vulkan developer would have to configure the layers either programmatically or by using environment variables specified by the layers documentation, which required a significant and continuous learning curve as the Vulkan layers capabilities evolved.

Vulkan Configurator was created to present the Vulkan layers with an intuitive interface enabling developers to use layer features with existing Vulkan applications, instantly and dramatically reducing development iteration time as no compilation, no learning of the new settings, and no tracking of the new features is required. The features are directly available in the GUI.

When we open *Vulkan Configurator*, we are greeted with the following window.



Vulkan Configurator 2.4.3 <ACTIVE>

Tools Help

Vulkan Layers Management

- ( ) Layers Fully Controlled by the Vulkan Applications
- (●) Overriding Layers by the Vulkan Configurator
  - ☐ Apply only to the Vulkan Applications List — Edit Applications...
  - ☐ Continue Overriding Layers on Exit

Vulkan Layers Configurations

- ( ) API dump
- ( ) Frame Capture
- ( ) Portability
- ( ) Synchronization
- (●) Validation

Vulkan Application Launcher

Application: vkcube

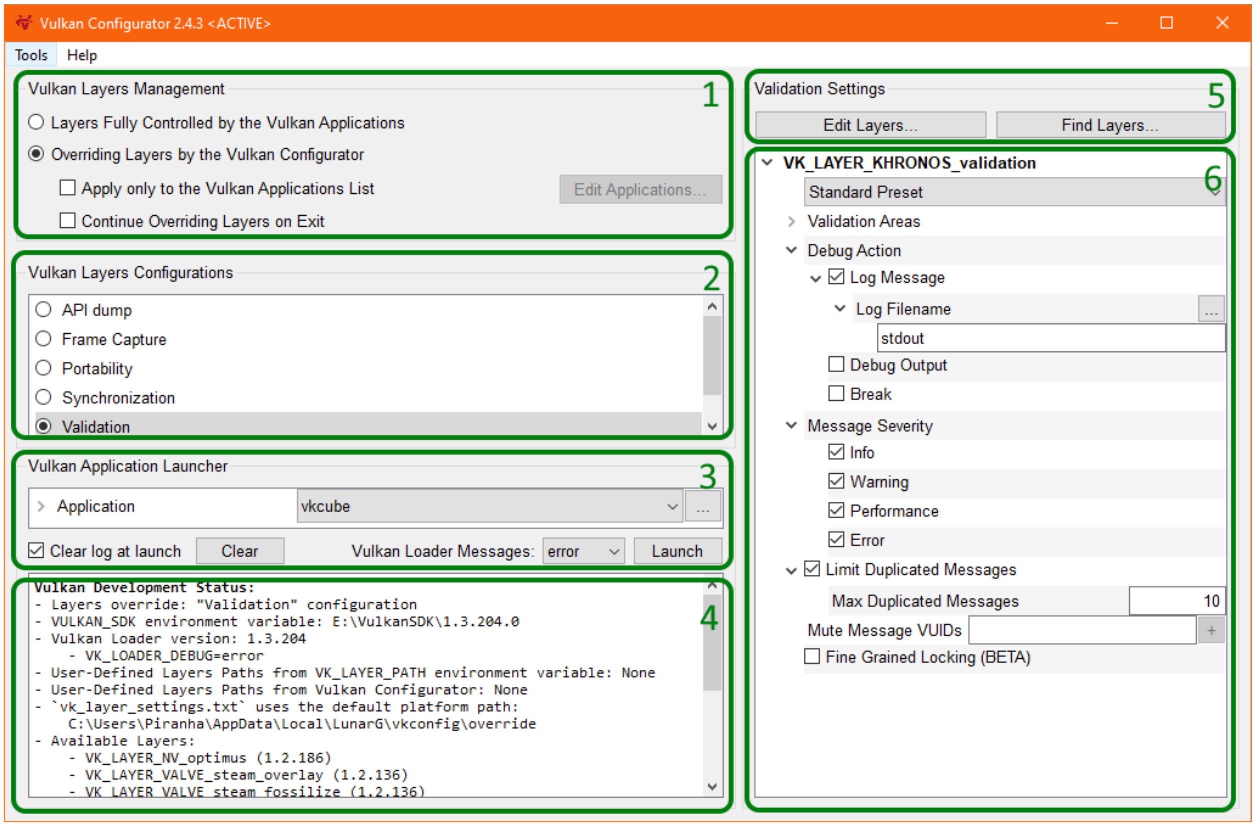
☑ Clear log at launch — Clear — Vulkan Loader Messages: error — Launch

```
Vulkan Development Status:
- Layers override: "Validation" configuration
- VULKAN_SDK environment variable: E:\VulkanSDK\1.3.204.0
- Vulkan Loader version: 1.3.204
    - VK_LOADER_DEBUG=error
- User-Defined Layers Paths from VK_LAYER_PATH environment variable: None
- User-Defined Layers Paths from Vulkan Configurator: None
- `vk_layer_settings.txt` uses the default platform path:
    C:\Users\Piranha\AppData\Local\LunarG\vkconfig\override
- Available Layers:
    - VK_LAYER_NV_optimus (1.2.186)
    - VK_LAYER_VALVE_steam_overlay (1.2.136)
    - VK_LAYER_VALVE_steam_fossilize (1.2.136)
```

Validation Settings

Edit Layers... — Find Layers...

- VK_LAYER_KHRONOS_validation
  - Standard Preset
  - Validation Areas
  - Debug Action
    - ☑ Log Message
      - Log Filename: stdout
    - ☐ Debug Output
    - ☐ Break
  - Message Severity
    - ☑ Info
    - ☑ Warning
    - ☑ Performance
    - ☑ Error
  - ☑ Limit Duplicated Messages
    - Max Duplicated Messages: 10
  - Mute Message VUIDs
  - ☐ Fine Grained Locking (BETA)



Vulkan Configurator 2.4.3 <ACTIVE>

Tools Help

1 **Vulkan Layers Management**

- ( ) Layers Fully Controlled by the Vulkan Applications
- (●) Overriding Layers by the Vulkan Configurator
  - ☐ Apply only to the Vulkan Applications List — Edit Applications...
  - ☐ Continue Overriding Layers on Exit

2 **Vulkan Layers Configurations**

- ( ) API dump
- ( ) Frame Capture
- ( ) Portability
- ( ) Synchronization
- (●) Validation

3 **Vulkan Application Launcher**

Application: vkcube

☑ Clear log at launch — Clear — Vulkan Loader Messages: error — Launch

4
```
Vulkan Development Status:
- Layers override: "Validation" configuration
- VULKAN_SDK environment variable: E:\VulkanSDK\1.3.204.0
- Vulkan Loader version: 1.3.204
    - VK_LOADER_DEBUG=error
- User-Defined Layers Paths from VK_LAYER_PATH environment variable: None
- User-Defined Layers Paths from Vulkan Configurator: None
- `vk_layer_settings.txt` uses the default platform path:
    C:\Users\Piranha\AppData\Local\LunarG\vkconfig\override
- Available Layers:
    - VK_LAYER_NV_optimus (1.2.186)
    - VK_LAYER_VALVE_steam_overlay (1.2.136)
    - VK_LAYER_VALVE_steam_fossilize (1.2.136)
```

5 **Validation Settings**

Edit Layers... — Find Layers...

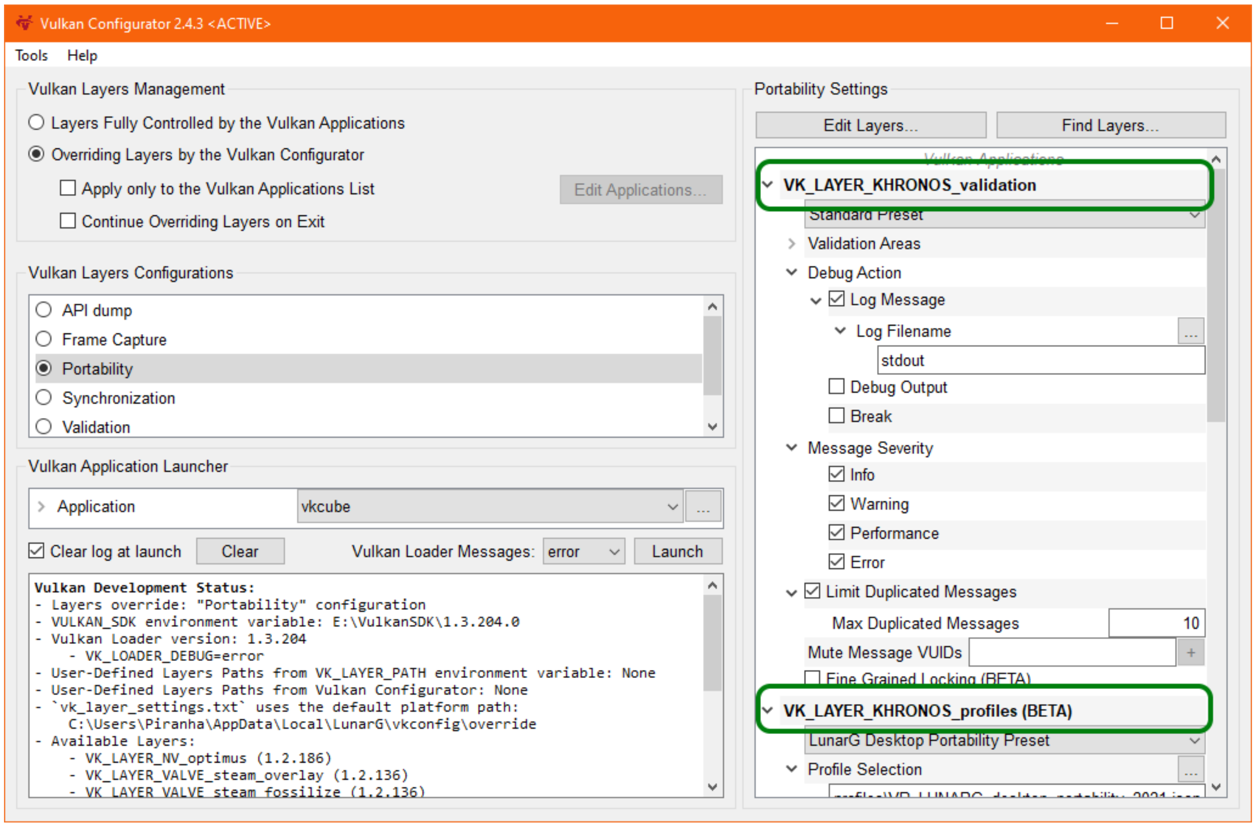
6
- VK_LAYER_KHRONOS_validation
  - Standard Preset
  - Validation Areas
  - Debug Action
    - ☑ Log Message
      - Log Filename: stdout
    - ☐ Debug Output
    - ☐ Break
  - Message Severity
    - ☑ Info
    - ☑ Warning
    - ☑ Performance
    - ☑ Error
  - ☑ Limit Duplicated Messages
    - Max Duplicated Messages: 10
  - Mute Message VUIDs
  - ☐ Fine Grained Locking (BETA)

The Vulkan Configurator UI comprises six areas:

1) Vulkan Layers Management: this area controls whether the Vulkan Layers override is active or not. It also determines whether the override is applied only to a selection of Vulkan applications or to all Vulkan applications. Finally, this area specifies whether the override remains active or not when Vulkan Configurator is closed.
2) Vulkan Layers Configurations: the list of pre-configured layers configurations. Vulkan Configurator is installed with a selection of built-in configurations that are listed on the screenshot. Each built-in configuration is designed to handle a specific Vulkan application developer use case. Using the context menu, we can design user-defined layer configurations to create layers configurations for our specific use cases.
3) Vulkan Application Launcher: this area allows running any Vulkan application with the selected layers configuration.
4) Log window: on start-up, when selecting a layer configuration or updating the layers list of a layer configuration, the log window will display the "Vulkan Development Status" which reports the version of various components, relevant paths for Vulkan developers, and the list of available layers. When launching a Vulkan application from Vulkan Configurator, the log window will display anything sent to stdout or stderr from the Vulkan layers, Vulkan applications, and the Vulkan Loader.
5) Layers selection for a layers configuration: the "Edit Layers…" button allows opening the "*Edit Vulkan Layers*" window to select the layers for the following actions:
    a) to override,
    b) to exclude,
    c) or to be handled by the Vulkan applications.

    The "*Find Layers...*" button allows adding paths to find additional layers on the system.
6) Layers configuration settings: the tree of settings for each layer. If the layers have setting presets, they are displayed just below the layer name.

Select the “Portability” built-in configuration from the “Vulkan Layers Configurations” list.



Vulkan Layers Management
- ○ Layers Fully Controlled by the Vulkan Applications
- ● Overriding Layers by the Vulkan Configurator
  - ☐ Apply only to the Vulkan Applications List
  - ☐ Continue Overriding Layers on Exit

Vulkan Layers Configurations
- ○ API dump
- ○ Frame Capture
- ● Portability
- ○ Synchronization
- ○ Validation

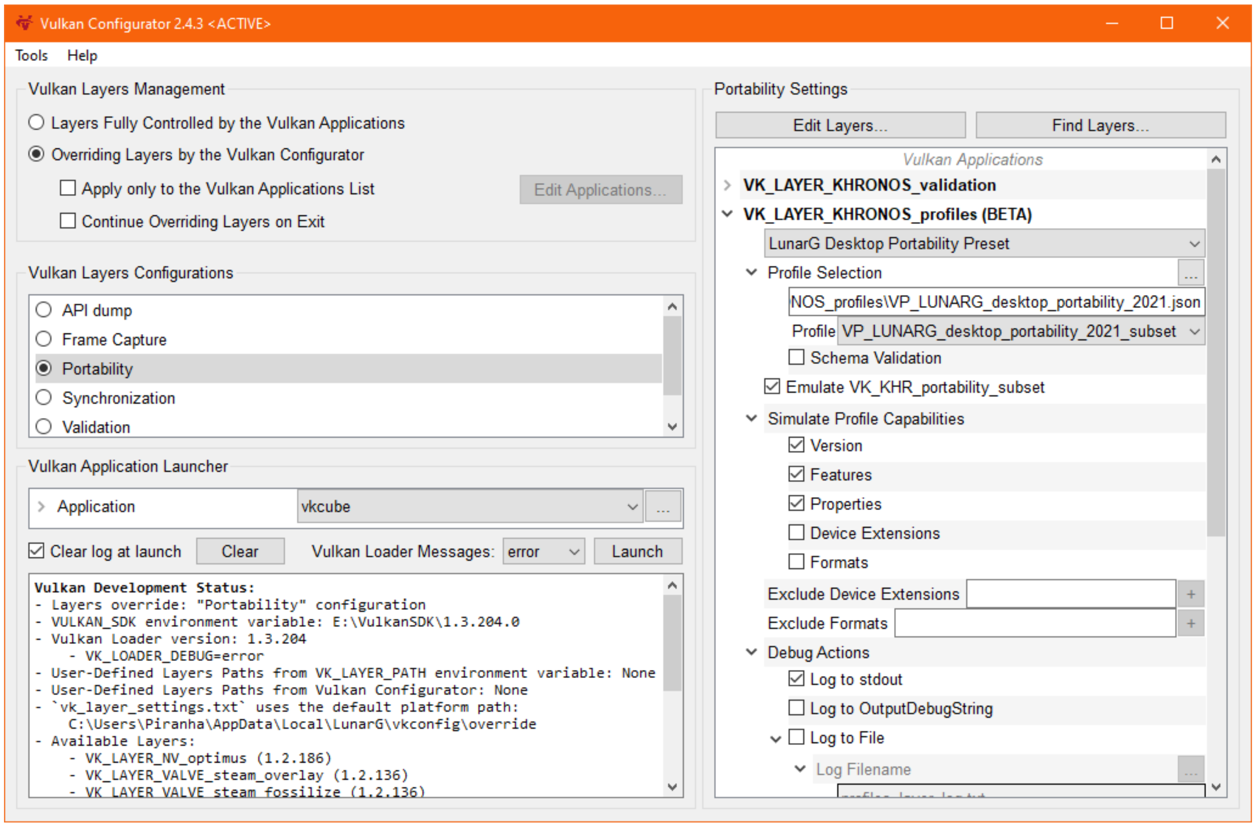
Vulkan Application Launcher
- Application: vkcube
- ☑ Clear log at launch
- Vulkan Loader Messages: error

```
Vulkan Development Status:
- Layers override: "Portability" configuration
- VULKAN_SDK environment variable: E:\VulkanSDK\1.3.204.0
- Vulkan Loader version: 1.3.204
    - VK_LOADER_DEBUG=error
- User-Defined Layers Paths from VK_LAYER_PATH environment variable: None
- User-Defined Layers Paths from Vulkan Configurator: None
- `vk_layer_settings.txt` uses the default platform path:
    C:\Users\Piranha\AppData\Local\LunarG\vkconfig\override
- Available Layers:
    - VK_LAYER_NV_optimus (1.2.186)
    - VK_LAYER_VALVE_steam_overlay (1.2.136)
    - VK_LAYER_VALVE_steam_fossilize (1.2.136)
```

Portability Settings

Vulkan Applications
- **VK_LAYER_KHRONOS_validation**
  - Standard Preset
  - Validation Areas
  - Debug Action
    - ☑ Log Message
      - Log Filename: stdout
    - ☐ Debug Output
    - ☐ Break
  - Message Severity
    - ☑ Info
    - ☑ Warning
    - ☑ Performance
    - ☑ Error
  - ☑ Limit Duplicated Messages
    - Max Duplicated Messages: 10
  - Mute Message VUIDs
  - ☐ Fine Grained Locking (BETA)
- **VK_LAYER_KHRONOS_profiles (BETA)**
  - LunarG Desktop Portability Preset
  - Profile Selection

This configuration includes the *Vulkan Validation layer* and the *Vulkan Profiles layer*.
To the right, we can see the layers settings.



Vulkan Layers Management
- ○ Layers Fully Controlled by the Vulkan Applications
- ● Overriding Layers by the Vulkan Configurator
  - ☐ Apply only to the Vulkan Applications List
  - ☐ Continue Overriding Layers on Exit

Vulkan Layers Configurations
- ○ API dump
- ○ Frame Capture
- ● Portability
- ○ Synchronization
- ○ Validation

Vulkan Application Launcher
- Application: vkcube
- ☑ Clear log at launch
- Vulkan Loader Messages: error

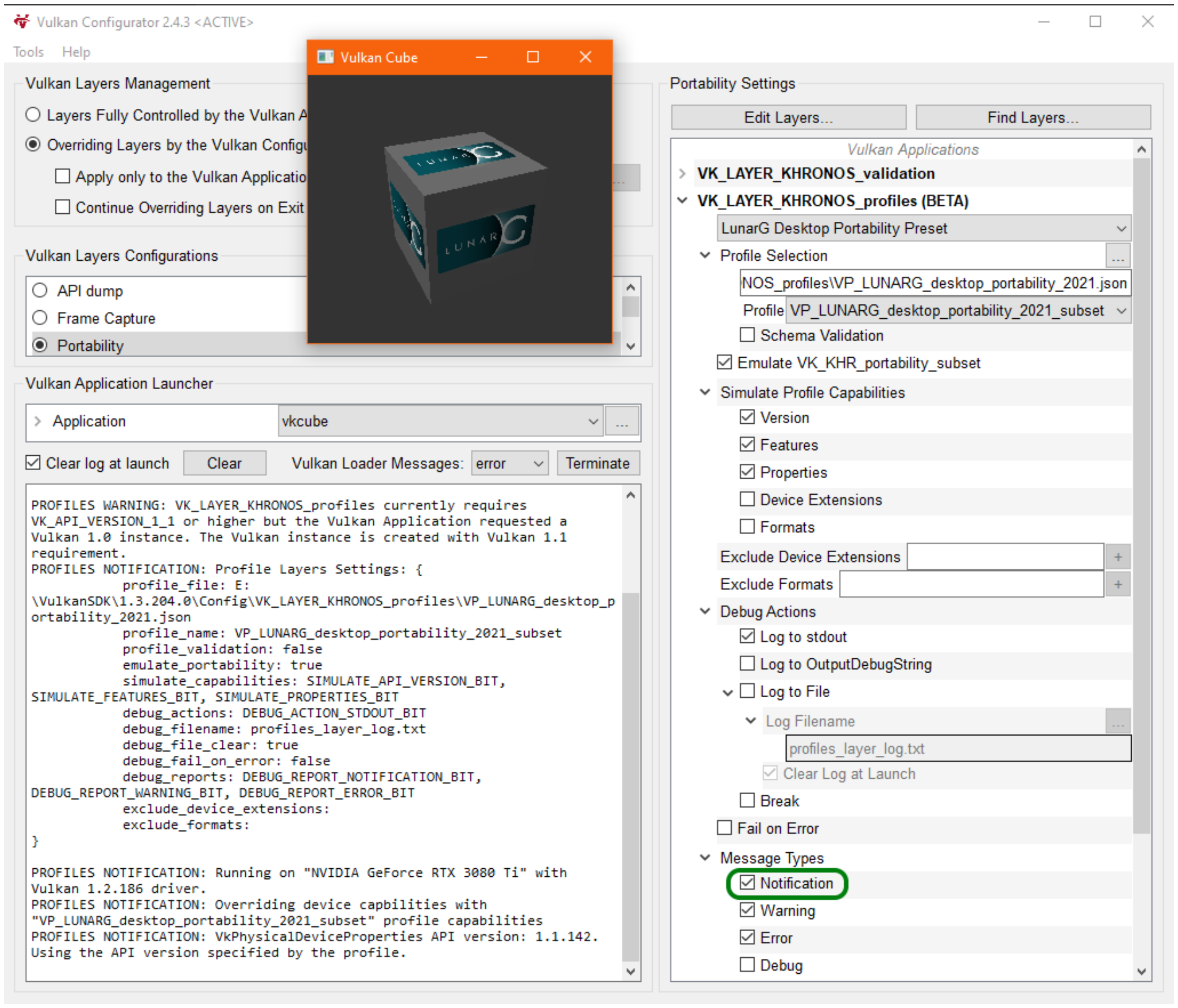
```
Vulkan Development Status:
- Layers override: "Portability" configuration
- VULKAN_SDK environment variable: E:\VulkanSDK\1.3.204.0
- Vulkan Loader version: 1.3.204
    - VK_LOADER_DEBUG=error
- User-Defined Layers Paths from VK_LAYER_PATH environment variable: None
- User-Defined Layers Paths from Vulkan Configurator: None
- `vk_layer_settings.txt` uses the default platform path:
    C:\Users\Piranha\AppData\Local\LunarG\vkconfig\override
- Available Layers:
    - VK_LAYER_NV_optimus (1.2.186)
    - VK_LAYER_VALVE_steam_overlay (1.2.136)
    - VK_LAYER_VALVE_steam_fossilize (1.2.136)
```

Portability Settings
- **VK_LAYER_KHRONOS_validation**
- **VK_LAYER_KHRONOS_profiles (BETA)**
  - LunarG Desktop Portability Preset
  - Profile Selection
    - NOS_profiles\VP_LUNARG_desktop_portability_2021.json
    - Profile: VP_LUNARG_desktop_portability_2021_subset
    - ☐ Schema Validation
  - ☑ Emulate VK_KHR_portability_subset
  - Simulate Profile Capabilities
    - ☑ Version
    - ☑ Features
    - ☑ Properties
    - ☐ Device Extensions
    - ☐ Formats
  - Exclude Device Extensions
  - Exclude Formats
  - Debug Actions
    - ☑ Log to stdout
    - ☐ Log to OutputDebugString
    - ☐ Log to File
      - Log Filename

We can hide the *Vulkan Validation layer* settings for now by clicking the carrot next to `VK_LAYER_KHRONOS_validation` and observe the *Vulkan Profiles layer* settings.

The Portability built-in configuration is created to be directly usable with the *LunarG Desktop Portability 2021* subset profile. Hence, we can check that `vkcube` can run correctly with this profile.

Selecting the *notification* message type, we get additional logging information to understand how the Vulkan Profiles behave.


Vulkan Configurator 2.4.3 <ACTIVE>
Tools
Help
Vulkan Cube
Vulkan Layers Management
Layers Fully Controlled by the Vulkan A
Overriding Layers by the Vulkan Config
Apply only to the Vulkan Applicatio
Continue Overriding Layers on Exit
Vulkan Layers Configurations
API dump
Frame Capture
Portability
Vulkan Application Launcher
Application
vkcube
Clear log at launch
Clear
Vulkan Loader Messages:
error
Terminate
Portability Settings
Edit Layers...
Find Layers...
Vulkan Applications
VK_LAYER_KHRONOS_validation
VK_LAYER_KHRONOS_profiles (BETA)
LunarG Desktop Portability Preset
Profile Selection
NOS_profiles\VP_LUNARG_desktop_portability_2021.json
Profile VP_LUNARG_desktop_portability_2021_subset
Schema Validation
Emulate VK_KHR_portability_subset
Simulate Profile Capabilities
Version
Features
Properties
Device Extensions
Formats
Exclude Device Extensions
Exclude Formats
Debug Actions
Log to stdout
Log to OutputDebugString
Log to File
Log Filename
profiles_layer_log.txt
Clear Log at Launch
Break
Fail on Error
Message Types
Notification
Warning
Error
Debug

```
PROFILES WARNING: VK_LAYER_KHRONOS_profiles currently requires
VK_API_VERSION_1_1 or higher but the Vulkan Application requested a
Vulkan 1.0 instance. The Vulkan instance is created with Vulkan 1.1
requirement.
PROFILES NOTIFICATION: Profile Layers Settings: {
           profile_file: E:
\VulkanSDK\1.3.204.0\Config\VK_LAYER_KHRONOS_profiles\VP_LUNARG_desktop_p
ortability_2021.json
           profile_name: VP_LUNARG_desktop_portability_2021_subset
           profile_validation: false
           emulate_portability: true
           simulate_capabilities: SIMULATE_API_VERSION_BIT,
SIMULATE_FEATURES_BIT, SIMULATE_PROPERTIES_BIT
           debug_actions: DEBUG_ACTION_STDOUT_BIT
           debug_filename: profiles_layer_log.txt
           debug_file_clear: true
           debug_fail_on_error: false
           debug_reports: DEBUG_REPORT_NOTIFICATION_BIT,
DEBUG_REPORT_WARNING_BIT, DEBUG_REPORT_ERROR_BIT
           exclude_device_extensions:
           exclude_formats:
}

PROFILES NOTIFICATION: Running on "NVIDIA GeForce RTX 3080 Ti" with
Vulkan 1.2.186 driver.
PROFILES NOTIFICATION: Overriding device capbilities with
"VP_LUNARG_desktop_portability_2021_subset" profile capabilities
PROFILES NOTIFICATION: VkPhysicalDeviceProperties API version: 1.1.142.
Using the API version specified by the profile.
```

For more information about the tool is available in the Vulkan Configurator documentation

Enabling the *Profiles layer* using environment variables

The Vulkan loader has environment variables for enabling layers.

These variables are `VK_LAYER_PATH` which is used to manually set the loader's search path for layers, and `VK_INSTANCE_LAYERS` which is used to set active layers and in what order they should be called.

The Profiles layer's name is `VK_LAYER_KHRONOS_profiles`. When turning it on, make sure it runs closest to the driver, since we want all layers and applications to see the simulated limits instead of the real limits.

Set the `VK_INSTANCE_LAYERS` environment variable to:
`VK_INSTANCE_LAYERS=<other layers>:VK_LAYER_KHRONOS_profiles`.

Next, we need to point the Vulkan Profiles layer to our desired Profile JSON file using the `VK_KHRONOS_PROFILES_FILENAME` environment variable by setting it to the path of the profiles JSON file.

From here, we can run our application and get the profile capabilities applied to the Vulkan device.

For additional information on layer settings, have a look at the Vulkan Profiles layer documentation.

Vulkan Profiles layer limitations

The *Vulkan Profiles layer* only changes the reported capabilities of a Vulkan driver. Used together with the Vulkan Validation layer, the solution will report errors when running Vulkan applications that use capabilities that are not reported by the layer.

However, the *Vulkan Profiles layer* has many limitations: it doesn't emulate Vulkan capabilities hence we are expecting it to be used on Vulkan developers machines which we think will effectively support more capabilities that the reality of the targeted devices in the Vulkan ecosystem by the Vulkan applications.

Furthermore, each operating system implements platform specific extensions. Hence, testing on all required operating systems remains necessary for full Vulkan capabilities requirements verification.

Also, it's not impossible that a Vulkan driver reports a supported capability but effectively this capability is not usable because of a Vulkan driver bug. The *Vulkan Profiles layer* will not help for such cases but we are hoping for *Vulkan Profile* definitions to help document such scenarios.

Finally, and probably most importantly, the *Vulkan Profiles layer* doesn't help with device performance testing but we are hoping using the Vulkan Profiles layer will free some development cycles for Vulkan developers to focus on this essential aspect.

Using the *Vulkan Profiles library*

The Vulkan Profiles library is a helper library for Vulkan application developers that provides the following set of APIs:

- APIs to verify instance-level and device-level support for a particular Vulkan profile.

- Instance and device creation APIs that automatically enable the extensions and features required by a particular Vulkan profile.
- Capability introspection APIs to query the extensions, features, properties, formats, and queue families required by a particular Vulkan profile.

A Vulkan sample using the Vulkan Profiles library is available in the Khronos Vulkan Samples repository.

Integration of the *Vulkan Profiles library* in an application

The Vulkan Profiles API library is provided as a header-only, C++ library (`vulkan/vulkan_profiles.hpp`) that is bundled with the Vulkan SDK. C++ applications thus can simply use the Vulkan Profiles library by including this header-only, C++ library with no Vulkan application build system changes.

The library is primarily designed to be dynamically linked to the Vulkan implementation (loader or ICD). If applications want to dynamically load Vulkan then they have to make sure (one way or another) that the Vulkan API symbols seen by the Vulkan Profiles header-only library are valid and correspond to the dynamically loaded entry points.

In order to enable support for other language bindings, the library is also available in a header + source pair (`vulkan_profiles.h` and `vulkan_profiles.cpp`). In the Vulkan Profiles repository, there is no build configuration for this variant of the library as it's not meant to be used as a standalone static or dynamic library. Instead, developers can drop the files into their own project to build the Vulkan profiles library into it. This may also come handy if the developer would like to optimize compilation times by not having to include the rather large header-only library in multiple source files.

The repository also contains a debug version of the Vulkan Profiles API library which allows logging unsupported capabilities of a Vulkan Profile when checking its support on a system.

The profile definitions are enabled depending on the preprocessor definitions coming from the Vulkan headers; thus the application has to make sure to configure the right set of preprocessor definitions. As an example, the `VP_ANDROID_baseline_2021` profile depends on the `VK_KHR_android_surface` instance extension; thus in order to use this profile, the application must define `VK_USE_PLATFORM_ANDROID_KHR`.

Generating Vulkan Profiles library

The *Vulkan Profiles library* is a header-only, C++ library. It doesn’t support loading dynamically Vulkan Profiles because the solution provides the simplest integration to an engine codebase. In most cases a Vulkan application can’t just load a profile, it implements that profile within the codebase. Hence, the Vulkan Profiles library just simplifies the initialization code of that implementation.

However, Vulkan developers may be interested in creating their own profiles, while still relying on the *Vulkan Profiles library*. For this reason, the *Vulkan Profiles library* can be trivially generated to include support for any desired Vulkan Profiles. The Vulkan developer just needs to clone the Vulkan Profiles repository and replace the list of profiles files before rebuilding the project and grabbing the regenerated `vulkan_profiles.hpp`.

Basic usage of the *Vulkan Profiles library*

The typically expected usage of the Vulkan Profiles library is for applications to target a specific profile with their application and leave it to the Vulkan Profiles library to enable any necessary extensions and features required by that profile.

In order to do so, the application first has to make sure that the Vulkan implementation supports the selected profile as follows:

```
VkResult result = VK_SUCCESS;
VkBool32 supported = VK_FALSE;
VpProfileProperties profile{
    VP_LUNARG_DESKTOP_PORTABILITY_2021_NAME,
    VP_LUNARG_DESKTOP_PORTABILITY_2021_SPEC_VERSION
};

result = vpGetInstanceProfileSupport(nullptr, &profile, &supported);
if (result != VK_SUCCESS) {
    // something went wrong
    ...
}
else if (supported != VK_TRUE) {
       // profile is not supported at the instance level
       ...
}
```

The above code example verifies the instance-level profile requirements of the `VP_LUNARG_desktop_portability_2021` profile, including required API version and instance extensions.

If the profile is supported by the Vulkan implementation at the instance level, then a Vulkan instance can be created with the instance extensions required by the profile as follows:

```
VkApplicationInfo vkAppInfo{ VK_STRUCTURE_TYPE_APPLICATION_INFO };

// Set API version to the minimum API version required by the profile
vkAppInfo.apiVersion = VP_LUNARG_DESKTOP_PORTABILITY_2021_MIN_API_VERSION;
VkInstanceCreateInfo vkCreateInfo{ VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO };
vkCreateInfo.pApplicationInfo = &vkAppInfo;
// set up your own instance creation parameters, except instance extensions
// as those will come from the profile
...

VpInstanceCreateInfo vpCreateInfo{};
createInfo.pCreateInfo = &vkCreateInfo;
createInfo.pProfile = &profile;

VkInstance instance = VK_NULL_HANDLE;
result = vpCreateInstance(&vpCreateInfo, nullptr, &instance);
if (result != VK_SUCCESS) {
    // something went wrong
    ...
}
```

The above code example will create a Vulkan instance with the API version and instance extensions required by the profile (unless the application overrides any of them, as explained later).

Make sure to set the `apiVersion` in the `VkApplicationInfo` structure at least to the minimum API version required by the profile, as seen above, to ensure the correct Vulkan API version is used.

Once a Vulkan instance is created, the application can check whether individual physical devices support the selected profile as follows:

```
result = vpGetPhysicalDeviceProfileSupport(instance, physicalDevice, &profile, &supported);
if (result != VK_SUCCESS) {
    // something went wrong
    ...
}
else if (supported != VK_TRUE) {
    // profile is not supported at the device level
    ...
}
```

Finally, once a physical device supporting the profile is selected, a Vulkan device can be created with the device extensions and features required by the profile as follows:

```
VkDeviceCreateInfo vkCreateInfo{ VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO };
// set up your own device creation parameters, except device extensions
// and device features as those will come from the profile
...

VpDeviceCreateInfo vpCreateInfo{};
createInfo.pCreateInfo = &vkCreateInfo;
createInfo.pProfile = &profile;

VkDevice device = VK_NULL_HANDLE;
result = vpCreateDevice(physicalDevice, &vpCreateInfo, nullptr, &device);
if (result != VK_SUCCESS) {
    // something went wrong
    ...
}
```

Advanced usage of the *Vulkan Profiles library*

The Vulkan Profiles library provides many functionalities to extend Vulkan Profiles with additional capabilities or even alter Vulkan Profiles to drop some requirements according to Vulkan developers needs. All these functionalities are described in the Vulkan Profiles library documentation.

For more information on the Vulkan Profiles Library API, have a look at the API reference.

Vulkan Profiles Toolset future improvements

The main area of possible improvements of the *Vulkan Profiles Toolset* is relative to the production of the *Vulkan Profiles*.

Vulkaninfo

In the short term, we are planning to add a Vulkan Profiles file export into `Vulkaninfo`. This would allow generating Device Vulkan Profiles directly from the console. Once this file is generated, we could simply modify it to remove specific capabilities to check that our Vulkan application fallback or fail correctly when these capabilities are not available on an end-user system.

Furthermore, it would be a good addition to provide a library form of `Vulkaninfo` so that any Vulkan tools could generate Vulkan Profiles files easily, for example to integrate the Vulkan capabilities of the system into a crash log.

Vulkan Capabilities minimum requirements

The Vulkan minimum requirements of the Vulkan specification are described in the features, limits and formats and additional capabilities sections of the specification. Unfortunately, there is no programmable way to access these minimum requirements. To enable this approach, vk.xml would need some additional information to reflect these requirements.

This would enable generating a Vulkan Profiles file with the Vulkan minimum requirements and provide a great base of an exhaustive representation of the Vulkan capabilities.

Generating Vulkan Profiles automatically

The LunarG Desktop Portability profile was created using an ad hoc approach. It serves at expressing a use case of the Vulkan Profiles but we should assume that most Vulkan applications should be able to produce their own profile representing the ecosystem of Vulkan devices and drivers the Vulkan applications should run on.

Such a Vulkan application profile could be generated by gathering all the profile files of all the Vulkan devices and with the oldest supported driver and using a Profiles Generator tool to combine the requirements of all these profiles. This tool remains to be created.

Finally, it would be useful to have a tool that compares Vulkan Profiles and shows the differences. For example, this tool could determine whether a Vulkan device exposes the Vulkan capabilities necessary for an engine represented by a Vulkan Profile without requiring access to this device.