Vulkan Layer Symbiosis within the Vulkan Ecosystem

Building an intuitive layer for Vulkan developers' usage

Christophe Riccio, LunarG
September 2021

Introduction 3

The Vulkan Loader, Vulkan Layers, and Vulkan Configurator Architecture 4
- Vulkan Configurator UI overview 4
- Vulkan Configurator UI Description 5
- Vulkan Layers Ecosystem Components Overview 7
- Vulkan Layers Components Standard System Locations 9
  - vk_layer_settings.txt 9
  - VkLayer_override.json 9

How layer settings are defined by the Vulkan Layer Manifest 11
- Layer Manifest Header 12
  - file_format_version 12
  - layer 12
  - name 13
  - api_version 13
  - implementation_version 13
  - description 14
  - introduction 14
  - url 14
  - platforms 14
  - status 15
- Layer Manifest Settings 15
  - key 16
  - label 16
  - description 16
  - type 16
  - url 17
  - platforms 17
  - status 17

view 17
expanded 18
dependence 18
settings 20
Layer Setting Types 20
BOOL: A boolean value 20
INT: A signed integer value 21
FLOAT: A floating point value 21
STRING: A string value 22
LOAD_FILE: A path to a file to load from 22
SAVE_FILE: A path to a file to save to 23
SAVE_FOLDER: A folder used to save files in it 23
ENUM: Selecting a single value among predefined values 24
FLAGS: Selecting multiple predefined values 25
LIST: A selectable list of strings and integers 26
FRAMES: A series of frames 27
GROUP: A settings container 27
Built-in variables for LOAD_FILE, SAVE_FILE and SAVE_FOLDER setting types 27
Settings Presets 27

How layer settings are accessed by Vulkan Layer code 30
The layer settings helper library 30
Layer settings helper library APIs 30
bool IsLayerSetting(const char *layer_name, const char *setting_key) 30
bool GetLayerSettingBool(const char *layer_name, const char *setting_key) 30
int GetLayerSettingInt(const char *layer_name, const char *setting_key) 30
double GetLayerSettingFloat(const char *layer_name, const char *setting_key) 30
std::string GetLayerSettingString(const char *layer_name, const char *setting_key) 30
Strings GetLayerSettingStrings(const char *layer_name, const char *setting_key) 31
List GetLayerSettingList(const char *layer_name, const char *setting_key) 31
std::string GetLayerSettingFrames(const char *layer_name, const char *setting_key) 31
void InitLayerSettingsLogCallback(LAYER_SETTING_LOG_CALLBACK callback) 31

Conclusion 32

References 33

Revisions 33

Introduction

Vulkan Layers are well known by Vulkan developers, largely due to the Validation layer that is an essential part of the Vulkan developers' daily tool kit.

Previously, using the Validation layer and customizing its features was done either programmatically or by using environment variables specified by the Validation layer documentation, which required a significant and continuous learning curve as the Vulkan ecosystem capabilities for developers evolved.

Vulkan Configurator was created to tackle this issue, presenting the Khronos and LunarG Vulkan layers with an intuitive interface enabling developers to use layer features with existing Vulkan applications, instantly and dramatically reducing development iteration time.

With the July 2021 release of the Vulkan SDK, we opened up the system that allows creating this efficient workflow so that any third-party Vulkan Layers in the ecosystem could leverage it and make their layers as easy to use as the Khronos and LunarG layers by the Vulkan application developers.

The purpose of this whitepaper is to describe how to create a Vulkan Layer in symbiosis with the Vulkan ecosystem. By this, we mean creating a layer that can benefit from the tools of the Vulkan ecosystem and follow the Vulkan ecosystem conventions so that Vulkan developers don’t have to keep up with constant new behaviors. Instead we can rely on the behavior they are used to and we can focus on using the features of the layers.

First we will give a quick overview of the Vulkan Configurator, Vulkan Loader, and Layers architecture. Then, we will focus particularly on the layer settings from two different aspects:

- How layer settings are defined by the Vulkan Layer manifest
- How layer settings are accessed by the Vulkan Layer code

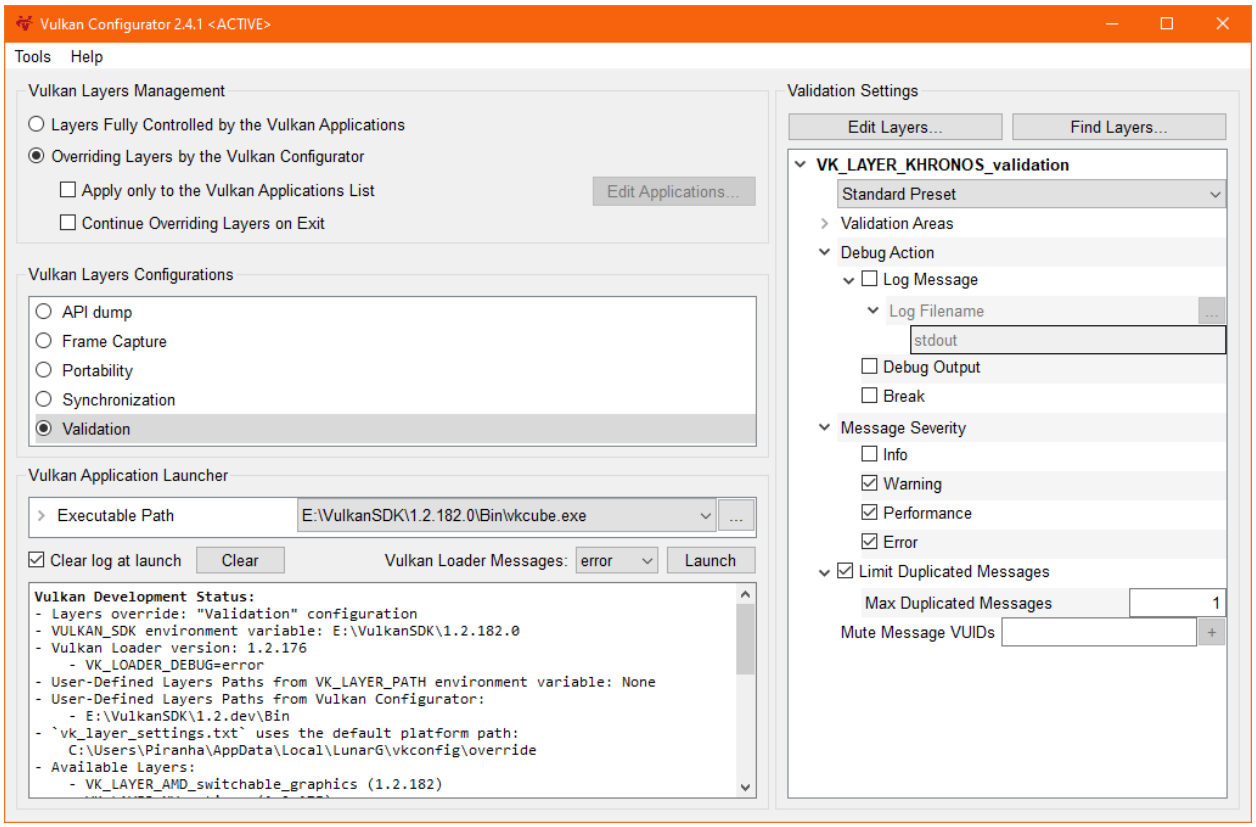
Finally, we will discuss future possible work to keep improving the Vulkan ecosystem using the Vulkan layers.

The Vulkan Loader, Vulkan Layers, and Vulkan Configurator Architecture

Vulkan Configurator UI overview

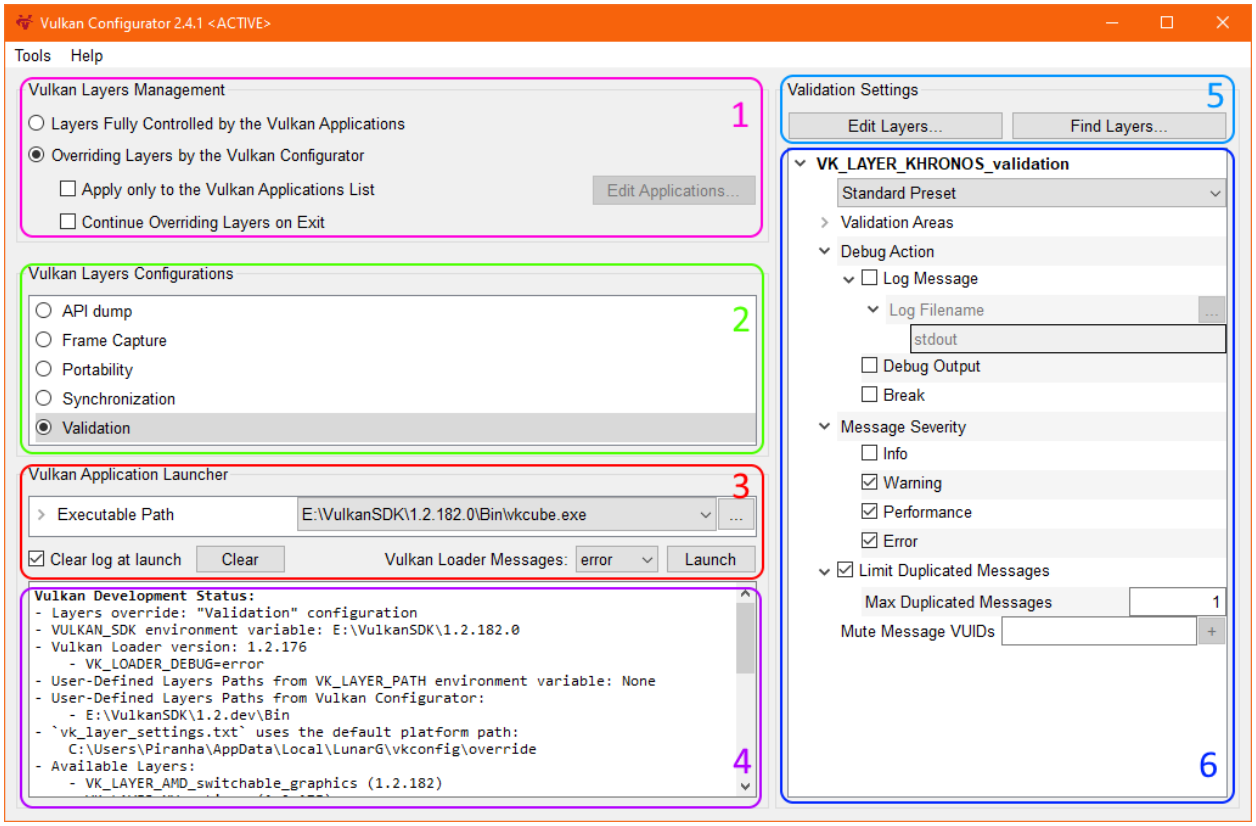
The new redesigned Vulkan Configurator was introduced with the August 2020 release of the Vulkan SDK to improve Vulkan application developers workflow, allowing developers to debug Vulkan applications with different modes quickly and with no learning curve. Additionally, as soon as a Vulkan layer implements a feature, the Vulkan application developers are exposed to the new features through the UI, without any need to scrutinize the Vulkan SDK release notes or the layer documentation.

Following is a screenshot of Vulkan Configurator the first time the application was launched by the Vulkan application developer:



By default, the environment is set up for standard validation of Vulkan applications.

Vulkan Configurator UI Description



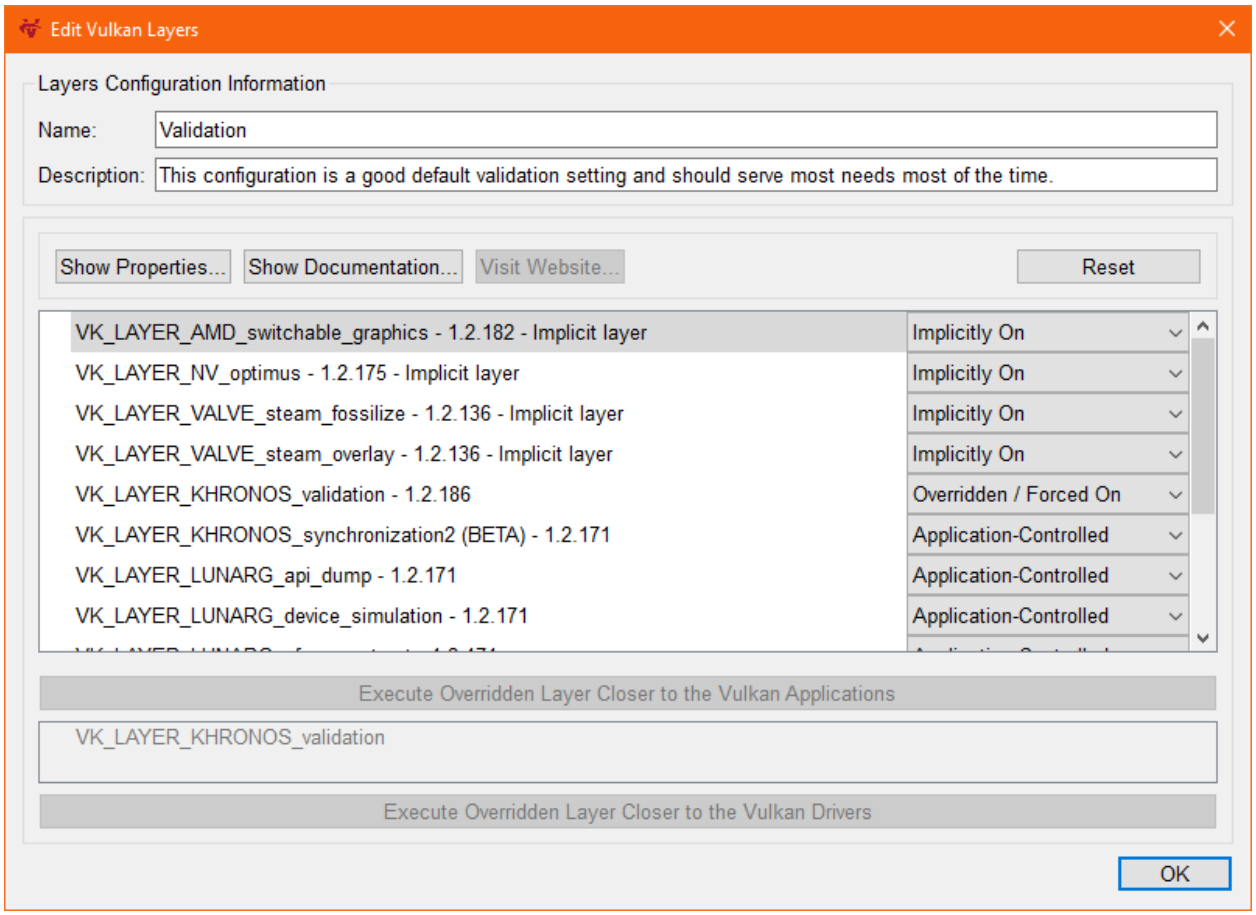
The Vulkan Configurator UI is comprised of six areas:

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 (screenshot below) 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.

   This area is the main subject of interest in this document; we will explore how to add these settings to a layer.



Edit Vulkan Layers window

For more information about the UI, a whitepaper and demo by Richard Wright are available.

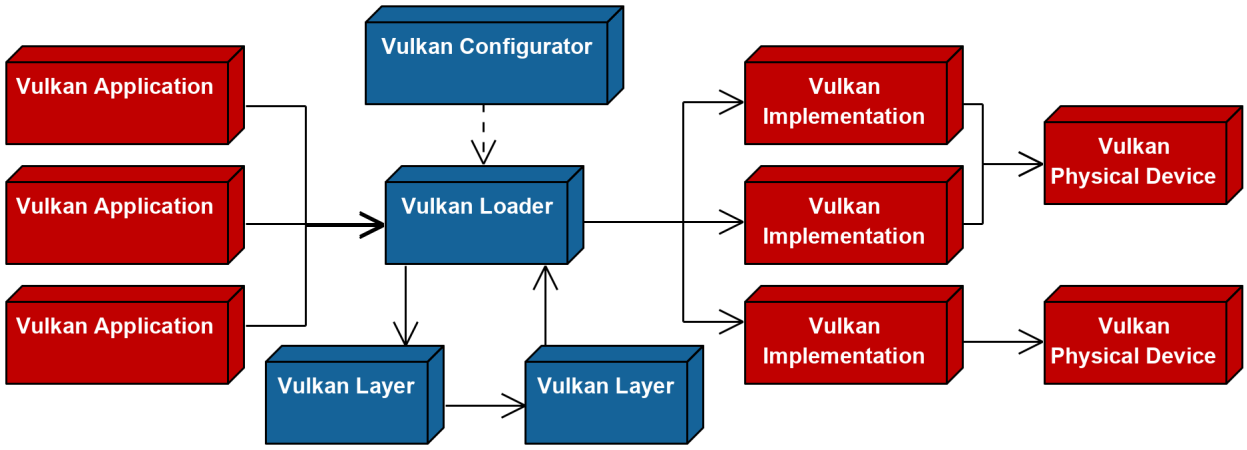
Vulkan Layers Ecosystem Components Overview

In this section, we will explore the different components that form the Vulkan Layers ecosystem and how they are connected to each other.

The Vulkan Loader and Vulkan Layers architecture is a thrilling topic. Sadly, it's not the focus of this whitepaper, but this topic was already largely discussed by Loader and Layer interface documentation and this excellent guide so we will remain at the overview level in this document.

The Vulkan loader is the central arbiter in the Vulkan runtime. The application talks directly to the loader and only to the loader, which then deals with enumerating and validating the layers requested, enumerating Vulkan implementations and organising them, and presenting a unified interface to the application.

Vulkan Layers are optional libraries that augment the Vulkan system. They can intercept, evaluate, and modify existing Vulkan functions on their way from the application down to the Vulkan Physical Device as shown on the following figure.



Vulkan API calls flow from the Vulkan applications to the Vulkan physical device

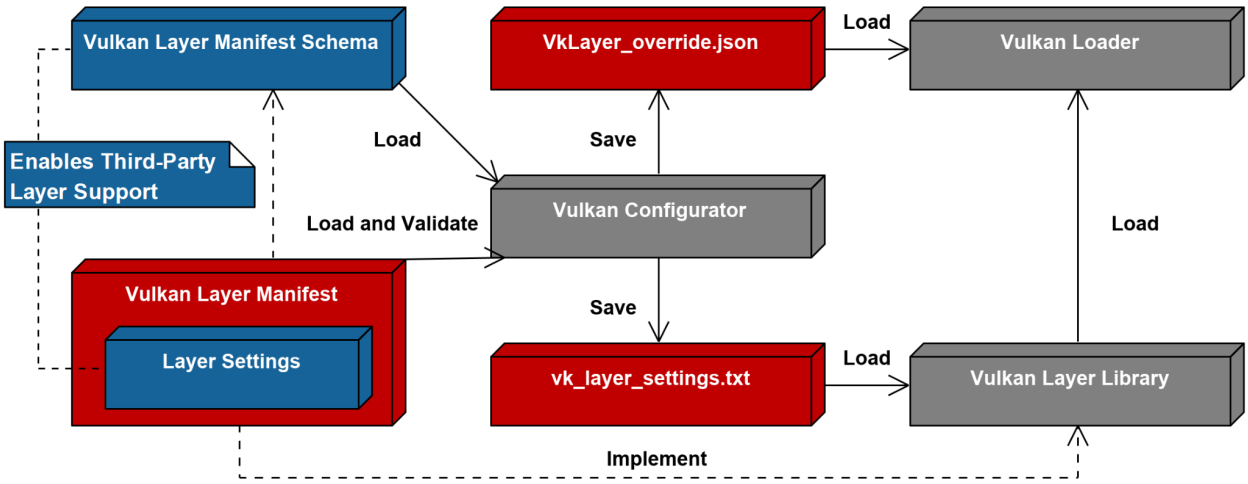
Each layer can choose to hook (intercept) any Vulkan functions that in turn can be ignored or augmented. A layer does not need to intercept all Vulkan functions. It may choose to intercept all known functions, or it may choose to intercept only one function.

The Vulkan Override Layer is an implicit meta-layer found on the system with the name `VK_LAYER_LUNARG_override` and controlled by the `VkLayer_override.json` file. It is the mechanism used by the Vulkan Configurator to override Vulkan layers. This layer contains:

- the ordered list of layers to activate
- the list of layers to exclude from execution
- the list of paths for executables to which the layers override applies. If this list is empty, the override is applied to every application upon startup.

Vulkan Configurator can be used to override the list of Vulkan Layers by generating the `VkLayer_override.json` file. The Vulkan Loader will dispatch Vulkan API calls based on the content of the `VkLayer_override.json` file.

Furthermore, Vulkan Configurator can control how the Vulkan Layers operate themselves by generating the `vk_layer_settings.txt` file that contains the layer settings. This file stores the setting values chosen by Vulkan developers through the Vulkan Configurator UI. The file is read by the Vulkan layers to figure out the Vulkan developers' requested behaviors. The definition of the layer settings is embedded in the Vulkan Layer manifest, a JSON file that can be validated using the layer manifest JSON schema.



Vulkan Layers Ecosystem Components interdependence

Vulkan Configurator is a tool that allows us to override the layer conveniently. However, the `VkLayer_override.json` and `vk_layer_settings.txt` files could be written manually or any Vulkan developer tool could integrate features to override Vulkan layers as well by generating the `VkLayer_override.json` and `vk_layer_settings.txt`.

These files only need to be placed at the correct locations so that the Vulkan Loader and the Vulkan Layers can find these files. This is discussed in the next section. Finally, the Vulkan Layers are also required to read the content of `vk_layer_settings.txt` correctly. As examples, the Khronos and LunarG layer settings are described here for VK_LAYER_KHRONOS_validation and here for other layers created by LunarG.

For these purposes, we provide a helper C++ Vulkan Layer settings library, discussed below in this document.

Vulkan Layers Components Standard System Locations

This section is useful for cases where we don't want to use the provided helper C++ Vulkan Layer settings library and instead want to implement a solution ourselves. This is not recommended as the helper library guarantees consistency across the Vulkan ecosystem.

`vk_layer_settings.txt`

The `vk_layer_settings.txt` file sets the Vulkan layers settings values that Vulkan developers want to use to configure the layers for a specific use-case.
The `vk_layer_settings.txt` file is located either globally at a system location or locally to the Vulkan application working directory. When the file is not found at the global system location, the Vulkan Layers should try to find the file locally.

The global system location that is platform specific:

- On Windows systems, the directory is found in the Windows registry from either the `HKEY_LOCAL_MACHINE` key or `HKEY_CURRENT_USER` key using the subkey "`Software\Khronos\Vulkan\Settings`".
- On Unix systems, the directory is "`~/.local/share/vulkan/settings.d`".

The global `vk_layer_settings.txt` path can be overridden using the environment variable `VK_LAYER_SETTINGS_PATH`.

`VkLayer_override.json`

The `VkLayer_override.json` file lists the Vulkan layers that the Vulkan Loader should load and the one that the Vulkan Loader should exclude -- even if the Vulkan applications request them.

The `VkLayer_override.json` file can only be set globally but the file itself can restrict the override of layers to only selected Vulkan applications.

The system location that is platform specific:

- On Windows systems, the directory is found in the Windows registry from either the `HKEY_LOCAL_MACHINE` key or `HKEY_CURRENT_USER` key using the subkey "`Software\Khronos\Vulkan\Settings`".
- On Unix systems, the directory is "`~/.local/share/vulkan/implicit_layer.d`".

Following is an example of `VkLayer_override.json` generated by Vulkan Configurator:

```
{
    "file_format_version": "1.1.2",
    "layer": {
        "api_version": "1.2.182",
        "app_keys": [
```

```
            "E:\\VulkanSDK\\1.2.188.0\\Bin\\vkcube.exe",
            "E:\\VulkanSDK\\1.2.188.0\\Bin\\vkcubepp.exe"
        ],
        "blacklisted_layers": [
        ],
        "component_layers": [
            "VK_LAYER_KHRONOS_validation"
        ],
        "description": "LunarG Override Layer",
        "disable_environment": {
            "DISABLE_VK_LAYER_LUNARG_override": "1"
        },
        "implementation_version": "1",
        "name": "VK_LAYER_LUNARG_override",
        "override_paths": [
        ],
        "type": "GLOBAL"
    }
}
```

- `app_keys` is optional and lists the path to the Vulkan applications that we want to override the layers. If `app_keys` is missing, then all Vulkan applications will have their layers overridden.
- `override_paths` is used for the Vulkan Loader to find additional layers or to replace layers located at standard system locations by the versions found at these paths.

How layer settings are defined by the Vulkan Layer Manifest

The layer manifest is a JSON file which contains the layer interface that is exposed to Vulkan developers and that allows configuring and documenting the Vulkan layers' functionalities.

The content of the manifest is specified with the layer manifest JSON schema. This document describes the version `"1.2.0"` of the JSON schema but the layer manifest file version history is available for more details.

Vulkan Configurator will systematically validate layers it's loading and exclude any layer whose manifest is not valid against the schema.

jsonschemavalidator.net can be used to develop the layer manifest file and ensure we write a valid file.

`VK_LAYER_KHRONOS_synchronization2` layer manifest as an example:

```
{
    "file_format_version": "1.2.0",
    "layer": {
        "name": "VK_LAYER_KHRONOS_synchronization2",
        "type": "GLOBAL",
        "library_path": "./libVkLayer_khronos_synchronization2.so",
        "api_version": "1.2.170",
        "implementation_version": "1",
        "description": "Khronos Synchronization2 layer",
        "introduction": "The VK_LAYER_KHRONOS_synchronization2 extension layer implements
the VK_KHR_synchronization2 extension.",
        "url":
"https://vulkan.lunarg.com/doc/sdk/latest/windows/synchronization2_layer.html",
        "platforms": [ "WINDOWS", "LINUX", "MACOS", "ANDROID" ],
        "status": "BETA",
        "device_extensions": [
            {
                "name": "VK_KHR_synchronization2",
                "spec_version": 1
            }
        ],
        "features": {
            "settings": [
                {
                    "key": "force_enable",
                    "env": "VK_SYNC2_FORCE_ENABLE",
                    "label": "Force Enable",
                    "description": "Force the layer to be active even if the underlying
driver already supports the synchonization2 extension.",
                    "type": "BOOL",
```

```
                    "default": false
                }
            ]
        }
    }
}
```

Many layer manifests are available and could be used as examples to write our layer manifest correctly:

- VK_LAYER_KHRONOS_validation manifest
- VK_LAYER_LUNARG_gfxreconstruct manifest
- VK_LAYER_LUNARG_api_dump manifest
- VK_LAYER_LUNARG_device_simulation manifest
- VK_LAYER_LUNARG_reference_1_2_0 manifest (Vulkan Configurator file for unit tests)

Layer Manifest Header

We are not going to describe all the layer manifest nodes as this work was already done in the Vulkan Loader documentation. However, we will focus our attention on nodes particularly relevant for the Vulkan Configurator as these nodes are exposed to Vulkan application developers by the GUI providing a form of in-application documentation which contributes to an intuitive usage of the Vulkan layers.

`file_format_version`

Manifest format major.minor.patch version number. Supported versions are: 1.0.0, 1.0.1, 1.1.0, 1.1.1, 1.1.2 and 1.2.0.

Version 1.2.0 is required to use layer settings.

The value must be a string that follows the syntax expressed by the regular expression `^[0-9]+\.[0-9]+\.[0-9]+$`.

Minimum Layer Manifest Version: 1.0.0
Presence of this node: Required

Examples:

- `"api_version": "1.1.2"`
- `"api_version": "1.2.0"`

`layer`

Defines the content of the layer manifest.

The `layer` node contains the header that consists of the following nodes: `name`, `api_version`, `implementation_version`, `description`, `introduction`, `url`, `platforms` and `status`.

Additional nodes exist and are documented by the loader and layer interface.

Minimum Layer Manifest Version: 1.0.0
Presence of this node: Required

`name`

The string used to uniquely identify this layer to applications.

The value must be a string that follows the syntax expressed by the regular expression `^VK_LAYER_[A-Z0-9]+_[A-Za-z0-9_]+.`

Minimum Layer Manifest Version: 1.0.0
Presence of this node: Required
Introspection Query: `vkEnumerateInstanceLayerProperties`

Examples:

```
- "name": "VK_LAYER_KHRONOS_validation"
- "name": "VK_LAYER_KHRONOS_synchronization2"
- "name": "VK_LAYER_LUNARG_gfxreconstruct"
```

`api_version`

The `major.minor.patch` version number of the Vulkan API that the shared library file for the library was built against.

Minimum Layer Manifest Version: 1.0.0
Presence of this node: Required
Introspection Query: `vkEnumerateInstanceLayerProperties`

Examples:

```
- "api_version": "1.0.33"
- "api_version": "1.1.130"
- "api_version": "1.2.176"
```

`implementation_version`

The version of the layer implemented. If the layer itself has any major changes, this number should change so the loader and/or application can identify it properly. This value is a complementary version number to `api_version`.

The value must be a string.

Minimum Layer Manifest Version: 1.0.0
Presence of this node: Required
Introspection Query: `vkEnumerateInstanceLayerProperties`

Examples:

- `"implementation_version": "36870"`
- `"implementation_version": "1.4.1"`
- `"implementation_version": "1"`
- `"implementation_version": "Build 176"`

`description`

The label of the layer.

The value must be a string.

Minimum Layer Manifest Version: 1.0.0
Presence of this node: Required
Introspection Query: `vkEnumerateInstanceLayerProperties`

Examples:

- `Khronos Validation Layer`
- `Khronos Synchronization2 layer`
- `LunarG device simulation layer`

`introduction`

A paragraph describing the layer and its intended use.

The value must be a string.

Minimum Layer Manifest Version: 1.2.0
Presence of this node: Optional

`url`

A link to the layer home page.

The value must be a string representing a valid accessible URL.

Minimum Layer Manifest Version: 1.2.0
Presence of this node: Optional

Examples:

- `"url": "https://vulkan.lunarg.com/doc/sdk/latest/windows/device_simulation_layer.html"`
- `"url": "https://vulkan.lunarg.com/doc/sdk/latest/windows/api_dump_layer.html"`
- `"url": "https://vulkan.lunarg.com/doc/sdk/latest/windows/khronos_validation_layer.html"`

`platforms`

The list of platforms supported by the layer.

The value must be an array of strings with the values among the following: `"WINDOWS"`, `"LINUX"`, `"MACOS"`, and `"ANDROID"`.

When the node is missing, the value is gathered from a parent node or set to support all platforms when no parent node has defined the node.

Minimum Layer Manifest Version: 1.2.0
Presence of this node: Optional

Examples:

```
- platforms: ["WINDOWS", "LINUX", "MACOS"]
- platforms: ["WINDOWS"]
```

`status`

The development status of the layer. It must be a value among the following: `"ALPHA"`, `"BETA"`, `"STABLE"`, or `"DEPRECATED"`.

When the node is missing, the value is gathered from a parent node or set to `"STABLE"` when no parent node has defined the node.

`"ALPHA"`: Typically refers to the stage where the layer is only available to the Layer developers for testing.
`"BETA"`: Typically refers to the stage where the layer is available to Vulkan developers but improvements remain necessary.
`"STABLE"`: Typically refers to the stage where the layer is ready for production.
`"DEPRECATED"`: The layer is no longer developed and may have been replaced by a different tool.

Minimum Layer Manifest Version: 1.2.0
Presence of this node: Optional

Examples:

```
- status: "ALPHA"
- status: "BETA"
```

Layer Manifest Settings

Layer settings were introduced with layer manifest version 1.2.0 and aimed mainly at providing a mechanism for configuring the layers using a graphical user interface. For this effect, LunarG provides Vulkan Configurator. Third-party graphical user interfaces could be created for the same purpose. The third-party interface would require a JSON parser capable of reading layer manifest version 1.2.0 and should know the details of the Vulkan Loader architecture.

The `settings` are stored in the `features` node of the `layer` node. It takes the form of an array of different setting types.

Each element of the `settings` node has required child nodes: `key`, `label`, `description` and `type`.

Example of layer settings definition in the layer manifest:

```
"features": {
    "settings": [
        {
            "key": "setting_bool_key",
            "label": "Key Bool Label",
            "description": "Key Bool Description.",
            "platforms": [ "WINDOWS", "LINUX", "MACOS" ],
            "status": "BETA",
            "view": "ADVANCED"
            "type": "BOOL",
            "default": false
        },
        {
            "key": "setting_string_key",
            "label": "Key String Label",
            "description": "Key String Description.",
            "type": "STRING",
            "default": "A String Default Value"
        }
    ]
}
```

Example `vk_layer_settings.txt` generated with the default values:

```
organization_name.setting_bool_key = false
organization_name.setting_string_key = A String Default Value
```

`key`

Identifier of the setting, for computational purposes.

`label`

Label of the setting, to expose the setting to Vulkan developers.

`description`

Explain the purpose of the setting.

`type`

The type of data stored by the setting, it must be a value among `BOOL`, `INT`, `FLOAT`, `STRING`, `LOAD_FILE`, `SAVE_FILE`, `SAVE_FOLDER`, `ENUM`, `FLAGS`, `LIST`, `FRAMES`, and `GROUP`. These setting types are described in detail by the Layer Setting Types section.

Furthermore, each element of the settings node had optional child nodes: `url`, `platforms`, `status`, `view`, `expanded`, `dependence`, and `settings`.

`url`

A URL for further documentation about the setting.

`platforms`

The list of platforms supported by the layer.

The value must be an array of strings with the values among the following: `"WINDOWS"`, `"LINUX"`, `"MACOS"`, and `"ANDROID"`.

When the node is missing, the value is gathered from a parent node or set to support all platforms when no parent node has defined the node.

Examples:

```
- platforms: ["WINDOWS", "LINUX", "MACOS"]
- platforms: ["WINDOWS"]
```

`status`

The development status of the layer setting. It must be a value among the following: `"ALPHA"`, `"BETA"`, `"STABLE"`, or `"DEPRECATED"`.

When the node is missing, the value is gathered from a parent node or set to `"STABLE"` when no parent node has defined the node.

`"ALPHA"`: Typically refers to the stage where the layer is only available to the Layer developers for testing.
`"BETA"`: Typically refers to the stage where the layer is available to Vulkan developers but improvements remain necessary.
`"STABLE"`: Typically refers to the stage where the layer is ready for production.
`"DEPRECATED"`: The layer is no longer developed and may have been replaced by a different tool.

Examples:

```
- status: "ALPHA"
- status: "BETA"
```

`view`

The level of visibility of a layer setting. It must be a value among the following: `"STANDARD"`, `"ADVANCED"`, or `"HIDDEN”`.

When the node is missing, the value is gathered from a parent node or set to `"STANDARD"` when no parent node has defined the node.

`"STANDARD"`: Settings directly visible to the Vulkan application developers.
`"ADVANCED"`: Advanced settings that should probably be used only by developers that have specific use-cases but probably would obfuscate common usages
`"HIDDEN"`: Settings that are hidden from the Vulkan application developers by the Vulkan layer developer.

Examples:

```
- view: "STANDARD"
- view: "HIDDEN"
```

expanded

Property that specifies whether by default the children of a setting should be visible or not.

Examples:

```
- expanded: true
- expanded: false
```

dependence

Multiple settings are often connected together and depend on the value of one so that the other setting becomes relevant. Expressing these dependencies help the Vulkan application developers to understand intuitively how the settings interact with each other.

The dependence node has two nodes: `"settings"` and `"mode."`

`"settings"` is an array of setting values to match.
`"mode"` is the matching operator mode and can either be set to one of the following values:

- `"none"`: None of the setting values should be satisfied to resolve the dependence.
- `"all"`: All the setting values must be satisfied to resolve the dependence.
- `"any"`: Any of the setting values must be satisfied to resolve the dependence.

Example of dependence between settings:

```
{
    "key": "use_spaces",
    "label": "Use Spaces",
    "description": "Setting this to true causes all tab characters to be replaced with
spaces",
    "type": "BOOL",
    "default": true,
    "settings": [
        {
            "key": "indent_size",
            "label": "Indent Size",
            "description": "Specifies the number of spaces that a tab is equal to",
            "type": "INT",
            "default": 4,
            "range": {
```

```
                "min": 1,
                "max": 16
            },
            "unit": "chars",
            "dependence": {
                "mode": "ALL",
                "settings": [
                    {
                        "key": "use_spaces",
                        "value": true
                    }
                ]
            }
        }
    ]
}
```

Example of dependence between a setting and a setting flag:

```
{
    "key": "memory_tracking_mode",
    "label": "Memory Tracking Mode",
    "description": "Specifies the memory tracking mode to use for detecting modifications to
mapped Vulkan memory objects. Available options are: page_guard, assisted, and unassisted.",
    "type": "ENUM",
    "flags": [
        {
            "key": "page_guard",
            "label": "page_guard",
            "description": "Tracks modifications to individual memory pages, which are
written to the capture file on calls to vkFlushMappedMemoryRanges, vkUnmapMemory, and
vkQueueSubmit. Tracking modifications requires allocating shadow memory for all mapped
memory."
        },
        {
            "key": "assisted",
            "label": "assisted",
            "description": "Expects the application to call vkFlushMappedMemoryRanges after
memory is modified; the memory ranges specified to the vkFlushMappedMemoryRanges call will
be written to the capture file during the call."
        }
    ],
    "default": "page_guard",
    "settings": [
     {
        "key": "page_guard_copy_on_map",
        "label": "Page Guard Copy on Map",
        "description": "When the page_guard memory tracking mode is enabled, copies the
content of the mapped memory to the shadow memory immediately after the memory is mapped.",
        "type": "BOOL",
        "default": true,
        "dependence": {
        "mode": "ALL",
        "settings": [
            {
                "key": "memory_tracking_mode",
                "value": "page_guard"
            }
        ]
```

```
    }
}
```

settings

A settings node can contain a settings node allowing a recursive structure of settings.

Example of layer settings definition with recursive `settings` within `settings`:

```
"settings": [
    {
        "key": "group",
        "type": "GROUP",
        "label": "Settings Group",
        "description": "Settings Group Description",
        "platforms": [ "WINDOWS", "MACOS" ],
        "status": "BETA",
        "view": "ADVANCED",
        "settings": [
            {
                "key": "int_inherit",
                "type": "INT",
                "label": "Integer Inherit",
                "description": "Integer Inherit Description",
                "default": 176
            },
            {
                "key": "int_override",
                "type": "INT",
                "label": "Integer Override",
                "description": "Integer Override Description",
                "platforms": [ "WINDOWS" ],
                "status": "ALPHA",
                "view": "HIDDEN",
                "default": 276
            }
        ]
    }
]
```

Layer Setting Types

BOOL: A boolean value

A setting that is either true or false.

Layer setting definition in the layer manifest:

```
{
    "key": "bool_key",
    "type": "BOOL",
    "label": "Bool Label",
    "description": "true or false",
    "default": true
}
```

Layer setting value initialization in vk_layer_settings.txt:

```
layer_name.bool_key = true
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_BOOL_KEY=true
```

`INT: A signed integer value`

A setting that represents an integer value. This setting type has optional nodes `"range"` and `"unit"`. Within the `"range"` node, the nodes `"min"` and `"max"` are optional too.

Layer setting definition in the layer manifest:

```
{
    "key": "int_key",
    "type": "INT",
    "label": "Int Label",
    "description": "Int Description",
    "default": 256,
    "range": {
        "min": 8,
        "max": 1024
    },
    "unit": "byte"

}
```

Layer setting value initialization in vk_layer_settings.txt:

```
layer_name.int_key = 256
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_INT_KEY=256
```

`FLOAT: A floating point value`

A setting that represents a floating-point precision value. This setting type has optional nodes `"range"` and `"unit"`. Within the `"range"` node, the nodes `"min"`, `"max"`, `"width"` and `"precision"` are optional too.

Layer setting definition in the layer manifest:

```
{
    "key": "float_key",
    "type": "FLOAT",
    "label": "Float Label",
    "description": "Float Description",
```

```
    "default": 76.5,
    "range": {
        "min": 75.1,
        "max": 82.2,
        "width": 2,
        "precision": 3
    },
    "unit": "second"
}
```

Layer setting value initialization in vk_layer_settings.txt:

```
layer_name.float_key = 76.500
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_FLOAT_KEY=76.500
```

`STRING: A string value`

A setting that is a string.

Layer setting definition in the layer manifest:

```
{
    "key": "string_key",
    "type": "STRING",
    "label": "String Label",
    "description": "String Description",
    "default": "A string default value"
}
```

Layer setting value initialization in vk_layer_settings.txt:

```
layer_name.string_key = A string default value
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_STRING_KEY="A string default value"
```

`LOAD_FILE: A path to a file to load from`

A setting that represents the path to a file to load. The setting may include the `"filter"` node to filter files with file dialogs.

The value may start with a PATH built-in variable.

Layer setting definition in the layer manifest:

```
{
```

```
    "key": "load_file_key",
    "type": "LOAD_FILE",
    "label": "Load File Label",
    "description": "Load File Description",
    "filter": "*.txt",
    "default": "${HOME}/layer_name/config.json"
}
```

Layer setting value initialization in vk_layer_settings.txt:

```
layer_name.load_file_key = C:\Users\Christophe\layer_name\config.json
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_LOAD_FILE_KEY=C:/Users/Christophe/layer_name/config.json
```

`SAVE_FILE: A path to a file to save to`

A setting that represents the path to save a file. The setting may include the `"filter"` node to filter files with file dialogs.

The value may contain a PATH built-in variable.

Layer setting definition in the layer manifest:

```
{
    "key": "save_file_key",
    "type": "SAVE_FILE",
    "label": "Save File Label",
    "description": "Save File Description",
    "filter": "*.txt",
    "default": "${HOME}/layer_name/log.txt"
}
```

Layer setting value initialization in vk_layer_settings.txt:

```
layer_name.save_file_key = C:\Users\Christophe\layer_name\log.txt
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_SAVE_FILE_KEY=C:/Users/Christophe/layer_name/log.txt
```

`SAVE_FOLDER: A folder used to save files in it`

A setting that represents the path to save files into a specified directory.

The value may contain a PATH built-in variable.

Layer setting definition in the layer manifest:

```
{
    "key": "save_folder_key",
    "type": "SAVE_FOLDER",
    "label": "Save Folder Label",
    "description": "Save Folder Description",
    "default": "${HOME}/layer_name"
}
```

Layer setting value initialization in vk_layer_settings.txt:

```
layer_name.save_folder_key = C:\Users\Christophe\layer_name
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_SAVE_FOLDER_KEY=C:/Users/Christophe/layer_name
```

`ENUM: Selecting a single value among predefined values`

A setting that represents a list of values which only one may be chosen. Each value is defined by an entry in the `"flags"` array. A `"flags"` value must define the `"key"`, `"label"` and `"description"` nodes. Optionally, a `"flags"` value may have additional nodes: `"url"`, `"platforms"`, `"status"`, `"view"`, `"expanded"` and `"settings"`.

Layer setting definition in the layer manifest:

```
{
    "key": "enum_key",
    "type": "ENUM",
    "label": "enum",
    "description": "enum case",
    "flags": [
        {
            "key": "flag0",
            "label": "Flag0",
            "description": "My flag0"
        },
        {
            "key": "flag1",
            "label": "Flag1",
            "description": "My flag1"
        },
        {
            "key": "flag2",
            "label": "Flag2",
            "description": "My flag2"
        }
    ],
    "default": "flag0"
}
```

Layer setting value initialization in vk_layer_settings.txt:

```
layer_name.enum_key = flag0
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_ENUM_KEY="flag0"
```

`FLAGS: Selecting multiple predefined values`

A setting that represents a list of values of which zero or more may be chosen. Each value is defined by an entry in the `"flags"` array. A `"flags"` value must define the `"key"`, `"label"` and `"description"` nodes. Optionally, a `"flags"` value may have additional nodes: `"url"`, `"platforms"`, `"status"`, `"view"`, `"expanded"` and `"settings"`.

Layer setting definition in the layer manifest:

```
{
    "key": "flags_key",
    "type": "FLAGS",
    "label": "flags",
    "description": "flags case",
    "flags": [
        {
            "key": "flag0",
            "label": "Flag0",
            "description": "My flag0"
        },
        {
            "key": "flag1",
            "label": "Flag1",
            "description": "My flag1"
        },
        {
            "key": "flag2",
            "label": "Flag2",
            "description": "My flag2"
        }
    ],
    "default": [ "flag0", "flag1" ]
}
```

Each element of `flags` may have the following additional properties: `url`, `status`, `view` and `platforms`.

Layer setting value initialization in vk_layer_settings.txt:

```
Layer_name.flags_key = flag0,flag1
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_FLAGS_KEY="flag0:flag1"
```

`LIST: A selectable list of strings and integers`

A list of values that may be integer or string data. The available values are listed in the `"list"` node. When `"list_only"` is true, the value set must be one included in the `"list"` node only.

Layer setting definition in the layer manifest:

```
{
    "key": "list_key",
    "type": "LIST",
    "label": "List Label",
    "description": "List Description",
    "list": [
        75,
        76,
        "A",
        "B",
        "C"
    ],
    "list_only": true,
    "default": [
        {
            "key": 76,
            "enabled": true
        },
        {
            "key": "A",
            "enabled": true
        },
        {
            "key": "B",
            "enabled": false
        }
    ]
}
```

Layer setting value initialization in [vk_layer_settings.txt]:

```
layer_name.list_key = 76,A
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_LIST_KEY="76:A"
```

FRAMES: A series of frames

A zero-based, comma-separated list of frames to output or a range of frames with a start and last separated by a dash. A value of -1 will output every frame. For example, "5-8" will output frame 5 until frame 8 inclusive.

Layer setting definition in the layer manifest:

```
{
    "key": "frames_key",
    "type": "FRAMES",
    "label": "Frames",
    "description": "Frames Description",
    "default": "76-82,75"
}
```

Layer setting value initialization in vk_layer_settings.txt:

```
layer_name.frames_key = 76-82,75
```

Layer setting value override with the `bash` console environment:

```
export VK_LAYER_NAME_FLAGS_KEY="76-82,75"
```

GROUP: A settings container

A setting to group multiple settings. This setting type is only really relevant when it includes a `"settings"` node. This setting doesn't output any value by itself.

Built-in variables for LOAD_FILE, SAVE_FILE and SAVE_FOLDER setting types

The path may start with a built-in variable which should be one of the following:

- `${HOME}`: The system user home directory
- `${VK_LOCAL}`: The default user accessible path for the data produced by the Vulkan SDK
- `${VK_APPDATA}`: The path to the hidden system directory used by the Vulkan SDK
- `${VULKAN_SDK}`: The path to the installed Vulkan SDK
- `${VULKAN_CONTENT}`: The path to the "content" directory in the Vulkan SDK

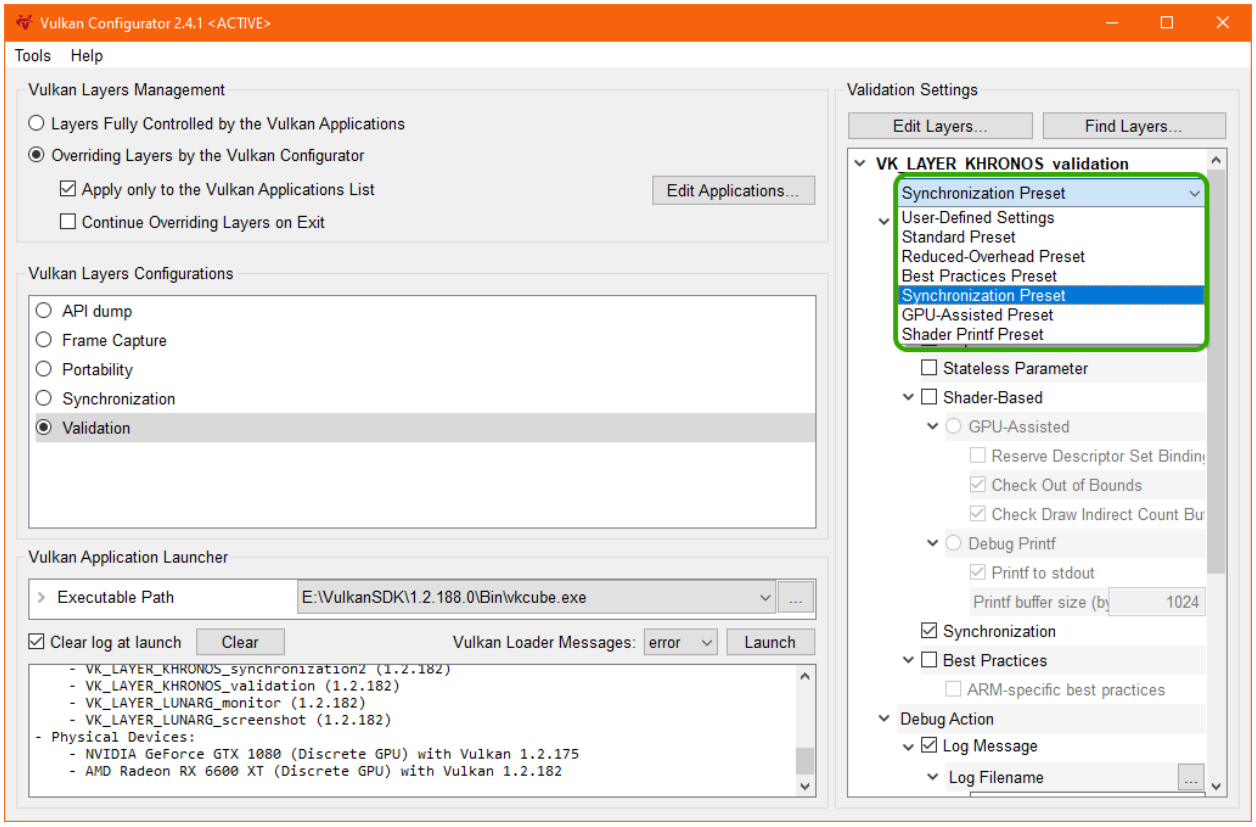
If a variable is used but is not one listed, the variable is not ignored and the string is unmodified.

Settings Presets

A preset is a collection of settings set to specific values. A preset typically represents a use-case that the Vulkan layer developer thinks is relevant.

A preset is stored as a value in the array of the `"presets"` node. Each value requires the `"label"`, `"description"` and `"settings"` nodes. Each value may have an `"url"` node, a `"platforms"` node and a `"status"` node.

The `"settings"` node in a preset is different from the other `"settings"` node. In the preset node, the settings are instantiations of the setting definitions. As a result, a preset setting only has a `"key"` and `"value"` node.



Example of preset with Validation layer presets exposed in Vulkan Configurator

```
"features": {
    "presets": [
        {
            "label": "Preset Inherit",
            "description": "Description Inherit",
            "settings": [
                {
                    "key": "int_required_only",
                    "value": 75
                }
            ]
        },
        {
            "label": "Preset Override",
            "description": "Description Override",
            "platforms": [ "WINDOWS", "MACOS" ],
            "status": "ALPHA",
            "settings": [
                {
                    "key": "int_required_only",
```

```
                    "value": 75
                }
            ]
        }
    ],
    "settings": [
        {
            "key": "int_required_only",
            "type": "INT",
            "label": "Integer",
            "description": "Integer Description",
            "default": 82
        }
    ]
}
```

How layer settings are accessed by Vulkan Layer code

The layer settings helper library

A helper library is provided to ensure the layer settings are loaded consistently with the rest of the Vulkan layer ecosystem.

This helper library is embodied in a simple header file "vk_layer_settings.h" and a simple source file "vk_layer_settings.cpp" that can be directly copied into the layer source.

The helper library handles:
- Finding the `vk_layer_settings.txt` file from all locations.
- Checking whether the layer settings are overridden by environment variables.
- Fetching the setting values, directly available to be used by the Vulkan layer.

Layer settings helper library APIs

`bool IsLayerSetting(const char *layer_name, const char *setting_key)`

Check whether the setting identified by `setting_key` of the layer identified by `layer_name` is set from `vk_layer_settings.txt` or an environment variable.

`bool GetLayerSettingBool(const char *layer_name, const char *setting_key)`

Query the setting value of the setting identified by `setting_key` of the layer identified by `layer_name`. The setting type must be BOOL otherwise an error will be emitted and the setting will be ignored.

`int GetLayerSettingInt(const char *layer_name, const char *setting_key)`

Query the setting value of the setting identified by `setting_key` of the layer identified by `layer_name`. The setting type must be INT otherwise an error will be emitted and the setting will be ignored.

`double GetLayerSettingFloat(const char *layer_name, const char *setting_key)`

Query the setting value of the setting identified by `setting_key` of the layer identified by `layer_name`. The setting type must be FLOAT otherwise an error will be emitted and the setting will be ignored.

`std::string GetLayerSettingString(const char *layer_name, const char *setting_key)`

Query the setting value of the setting identified by `setting_key` of the layer identified by `layer_name`. The setting type must be STRING, LOAD_FILE, SAVE_FILE, SAVE_FOLDER or ENUM otherwise an error will be emitted and the setting will be ignored.

`Strings GetLayerSettingStrings(const char *layer_name, const char *setting_key)`

Query the setting value of the setting identified by `setting_key` of the layer identified by `layer_name`. The setting type must be FLAGS otherwise an error will be emitted and the setting will be ignored.

`Strings` is defined as:

```
typedef std::vector<std::string> Strings;
```

`List GetLayerSettingList(const char *layer_name, const char *setting_key)`

Query the setting value of the setting identified by `setting_key` of the layer identified by `layer_name`. The setting type must be LIST.

`List` is defined as:

```
typedef std::vector<std::pair<std::string, int>> List;
```

`std::string GetLayerSettingFrames(const char *layer_name, const char *setting_key)`

Query the setting value of the setting identified by `setting_key` of the layer identified by `layer_name`. The setting type must be FRAMES.

`void InitLayerSettingsLogCallback(LAYER_SETTING_LOG_CALLBACK callback)`

Initialize the callback function to get error messages. By default the error messages are output to `stdout`. Use `nullptr` to return to the default behavior.

`LAYER_SETTING_LOG_CALLBACK` is defined by:

```
typedef void *(*LAYER_SETTING_LOG_CALLBACK)(const char *setting_key, const char
*message);
```

Conclusion

In this document, we saw an overview of the different components of the Vulkan layer ecosystem and how they connect together to make it intuitive for Vulkan application developers to leverage the layer features effectively.

There are more developments that could be made in the future to improve the Vulkan layers ecosystem:

- A Vulkan API could be created to pass the layer settings data to the layer instead of using `vk_layer_settings.txt`.
- Vulkan Configurator could be made into a C++ library so that it could be integrated in third-party tools to configure layers.
- Vulkan Configurator could handle `vk_layer_settings.txt` located next to the Vulkan application.
- Vulkan Configurator could handle more features of the layer manifest such as the `disable_environment` and `enable_environment` nodes.
- New settings could be added to make common usages consistent: log system, input system.
- The layer manifest schema could cover all the features of the layer manifest.

More ideas are welcome; if you encounter any issue, you can create a GitHub issue.

References

- Introducing the new Vulkan Configurator (2020)
- Introducing the new vkconfig, Vulkan Configurator (2020)
- Vulkan Ecosystem Enhancements SIGGRAPH 2021 presentation
- The Layer Settings helper library: vk_layer_settings.cpp and vk_layer_settings.h
- The Layer manifest schema
- Examples of layer manifests for all setting types
    - To make settings of any types
    - To make setting presets
    - To make setting groups
- The State of Vulkan on Apple Devices (2021)

Revisions

1. Initial document representing the design shipping with the September 2021 Vulkan SDK.